Abstract

Title of dissertation USING JOIN NETWORKS TO COMPUTE SATISFIABILITY

Carl Floyd Andersen, Doctor of Philosophy, 2008

Directed by Professor William Gasarch
Department of Computer Science

Satisfiability (SAT) of propositional logic formulas is a canonical NP-complete problem; algorithms for its solution have been studied for over forty years. The representational power of propositional logic allows a host of research and real-world problems to be solved using SAT solvers: some prominent areas of application include mathematics, circuit design and verification, and AI planning.

One technique receiving increasing recent attention is the use of quantified representations for SAT problems. Quantified representations are often more intuitive to use, can require exponentially less space, and in many cases allow speedup through their elimination of isomorphism. This dissertation explores the use of networks of joins operating upon quantified representations to compute key solver functions, including unit propagation and literal choice. The complexity of computing these functions using join networks becomes dependent upon the size of the truth assignment, or potential model explored at a given search space node. Because models are relatively small for many problems of interest, we show efficiency gains in these cases. JOINSAT, the implementation of these ideas, is competitive in performance with a number of state of the art satisfiability solvers.

USING JOIN NETWORKS TO COMPUTE SATISFIABILITY

by

Carl Floyd Andersen

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

To Alyson, who never gave up on me.

Acknowledgments

First, second and third I thank my wife Alyson Steele, who endured so much more than any young spouse and mother should have to. Through many years of uncertainty, doubt and privation she kept a cheerful heart and never unleashed upon me the resentment and frustration that she by all rights could have. Honey, the days ahead will be easier for all of us. I love you.

Next, I thank my parents, Kenneth and Lucille Andersen, who gave me years of advice and support as well as very significant financial help. In particular, I am grateful for the month I was writing my dissertation, when my mother and then my father came out to help care for our children, Jesse and Audrey, and drive them to their various schools. Mom and Dad, thank you so much, I absolutely could not have finished the dissertation or the Ph.D. without your help. But think of it this way: at least you are alive to see me finish! :)

I would like to thank other people in more or less chronological order. I thank my first advisor, Don Perlis, for exposing me to fascinating currents of research that promise to inform my work for years. I thank my office mates, Khemdut Purang and Darsana Purushothaman Josyula, for many hours of camaraderie and fascinating discussion about A.I. as well as the events of our times. I very much enjoyed your company. I thank my second advisor, Jeff Horty, for exposing me to a rigorous vision of research and of scientific writing. To whatever degree that care and precision are on display in the present document, it is thanks to Jeff's patient yet exacting tutelage. On a different note, I thank Felicia Chelliah for her welcoming smile over the long years: she was always glad to see me and hear about my family even after months of not seeing me at her office door.

I especially thank my present advisor, Bill Gasarch, for helping me formalize my dissertation vision and finally finish. Bill was always a fount of jokes, encouragement and good cheer and every session with him improved my morale. It was awfully good of him to take me under his wing and I won't ever forget it.

I thank the members of my dissertation committee for reading my dissertation and giving me helpful comments, particularly Dana Nau, who helped me improve my experiments.

Finally, I thank Geoff Sutcliffe of the University of Miami Computer Science Dept for allowing me to use his TPTPWorld system to run many hundreds of test runs quickly and easily. I shudder to think of how much time the testing would have taken me without his help.

Stephen had now lost count of time. It seemed to pass round him, or over him, in a perpetual muddle and hurry, or at least with so many things going on at once that he could not keep track of them, though he was aware that some guiding intelligence directed the obscure movements in the darkness. The only thing that was clear in his mind - the centre of his physical and mental activity except on those occasions when he was called away to dress a wound - was the pump, and the plain, urgent task of heaving it round so that the ship should not sink.

Hours passed. The pump was repaired and the midshipman in charge of it roused them all out. Another spell, and the heaving soon became mechanical again, the wind and the rain hardly noticed. Relief: deep and apparently momentary sleep: and they were called out again.

But [Jack] jerked into consciousness when the relief was called, and returned through the darkness to the starboard pump on Bonden's arm. There were fewer men at their duty now - more and more were hiding - and these worked silently, with much less strength: hope was fading, if it was not entirely dead. He called out 'Huzzay, heave round,' mechanically, and as he did he forced his mind to work out fresh ways of coming at the leak, and of steering the ship once it was stopped; Pakenham had made a rudder from spare topmasts...

'Thursday, 25 December. Course estimated E 10°S. Latitude observed 46°37′S. Longitude estimated 50°15′E. Winds light and variable with haze and rain. Sea calm with several small blocks of ice. PM hauled up foresail, veered out stop-water to check ship's way, and passed fothering-sail forward from abaft the sternpost, bowsing it taut from the fashion-pieces to the mizzen-chains. The sail answered and the pumps gained five foot in the day.'

-excerpts from Desolation Island, by Patrick O'Brian

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Importance of Satisfiability

The general problem of satisfiability, that is, the finding of models for logical theories, is an important research area in the field of automated deduction. The problem has a large number of academic and real-world applications, including AI Planning [37, 38, 64], circuit design [31], circuit verification [14, 59], software design [35], and even mathematics [43, 65], among others.

The best-known variety of satisfiability problem is SAT, the finding of models for propositional theories. SAT is a canonical NP-complete problem: not only can all NP-complete problems be translated to it in polynomial time, but its very simple form often makes these translations particularly concise. Among other problems of logical satisfiability, SAT is also noteworthy because the majority of research into satisfiability solvers has been directed at SAT and because most real-world satisfiability applications use SAT solvers. However, in recent years increasing attention has been directed toward the problem of quantified satisfiability, in which the problem theories include quantified variables. Most problems of interest can be represented either as SAT problems or as quantified problems, so the hope is to develop quantified solvers that outperform SAT solvers on the same "problem". This hope is justified by the fact that quantified solvers avoid the intrinsically exponential instantiation approach of SAT solvers.

This dissertation introduces a new algorithm for computing the class of quantified satisfiability problems in which the logical universe is held to be of a given finite size *n*. This new method maintains the satisfiability problem state as a network of joins akin to database joins. The method is competitive in speed with the state of the art algorithm, in which the quantified representation is translated to a ground (propositional) representation and a SAT solver is used to solve the resulting problem.

1.2 An Overview of the Dissertation

This dissertation's introductory chapter first examines the SAT problem and state of the art methods for its solution, including the currently dominant method, the DPLL algorithm. The chapter then discusses the problem of subsearch (section 1.5), or the maintenance at every DPLL iteration of crucial state data supporting the ongoing search for a logical model. DPLL-style solvers invariably use an inefficient subsearch representation and algorithm; I seek to improve upon both of these in the dissertation.

The introductory chapter next presents a new (quantified) satisfiability problem formalism, FQSAT (section 1.6). I believe that most SAT problems are initially specified as FQSAT problems by their authors; these problems are then translated to SAT and solved using variants of DPLL. The remainder of the chapter presents (in section 1.7) a solver exemplifying the translation-to-SAT approach, the MACE2 solver. MACE2 automatically translates FQSAT problems into SAT and then uses a DPLL-style solver. Since I believe that solving the original FQSAT problem directly is a more efficient approach, MACE2 is used in subsequent chapters as an explanatory counterpoint to JOINSAT, my own FQSAT solver.

In the second chapter, I discuss JOINSAT, the main research effort of the dissertation. This requires a description of past research into A.I. Production Systems, whose methods I adapt to satisfiability solvers (section 2.1). In sections 2.2-2.5, I discuss my main algorithmic efforts at length. Section 2.2 introduces the idea of a join network, and section 2.3 measures its formal complexity and contrasts it with that of MACE2. In sections 2.4-2.5 I describe the JOINSAT algorithm in full detail and with examples. The rest of the chapter summarizes the comparative algorithmic advantages for MACE2 and JOINSAT.

The third chapter discusses extensive optimizations made to the basic JOINSAT algorithm in an effort to increase its speed, as well as experimental testing comparing each variant's performance. These include completely redesigning its core representation twice, once to use precomputed information (section 3.3) and once to incorporate a more efficient Production Systems representation (section 3.4). A third optimization explores changing the structure of the join network in the hope of reducing the overall number of matches (section 3.5) Two other sections (3.6 and 3.7) discuss further representational changes which substantially improve overall performance. A final section (3.8) explores the use of static analysis techniques to compute the optimal join network organization.

The fourth chapter presents experimental testing of JOINSAT against five other state of the art FQSAT solvers, including MACE2. The chapter also presents the methodology used in these tests and discusses the results.

The fifth chapter examines related work, including solvers that operate on a quantified satisfiability problem that is more expressive than FQSAT.

The concluding chapter offers some ideas for future work and a summation of the research and its value.

1.3 A Brief Review of SAT Research

In this section, I present the SAT problem and briefly summarize salient issues in SAT research.

1.3.1 The Ground Satisfiability (SAT) Problem

SAT is the problem of satisfiability for finite theories of propositional logic.

Problem. Propositional Satisfiability (SAT). We are given a Boolean formula F, expressed as a set C of CNF clauses. Let the set of clause literals contained in the formula be $l_1,\dots,l_m$ and the set of variables $v_1,\dots,v_n$, where each clause literal l_i is either some variable v_j or its negation $\neg v_j$. Return either:

1. a model assignment mapping each variable to one of {TRUE,FALSE} that makes formula F true; or
2. FAILURE, if no such assignment exists.

SAT is a canonical NP-complete problem, in that literally scores of well-known NP-complete problems (e.g. traveling salesman, constraint satisfaction problem, 0-1 integer programming, to name just a few) can be easily reduced to it. Algorithms for its solution have been studied for over forty years, but the last fifteen have been particularly exciting: solver performance has improved by orders of magnitude, and translation to SAT has become a competitive option in several specialized domains, such as AI Planning. In the Deterministic division of the most recent International Planning Comptetition [1], two systematic SAT planners won top honors (SATPLAN [36] and MAXPLAN [12]).

1.3.2 SAT Research

An extensive survey [30] of algorithms for the SAT problem organizes them into two useful dimensions: whether the algorithm uses a discrete or a continuous representation, and whether the algorithm uses a set of hard constraints that must be satisfied, or instead tries to minimize some objective function. The SAT literature is so voluminous that I discuss only the most representative instances of each type of algorithm.

A SAT problem is specified as a set of discrete variables and constraints: a continuous algorithm converts these into continuous equations such that solutions to the equation correspond to binary solutions to the original discrete problem. For example, a variety of continuous algorithms convert SAT into an integer programming (IP) problem and then try to to solve it using a variety of linear programming techniques, e.g. branch and bound [10] and cutting planes [32], among others. Continuous methods have suffered in performance comparisons with discrete methods in recent years.

Discrete algorithms (often referred to as *ground* solvers in contrast to algorithms using quantified problem representations) retain the original discrete representation of the problem variables and clauses and search through the space of possible variable assignments until a solution is found or the search space is exhausted, indicating failure. Discrete unconstrained algorithms typically use some sort of local search through the assignment space. One of the earliest of these algorithms is GSAT [56], which uses a greedy local search and a heuristic of minimizing the number of unsatisfied clauses.

An ongoing problem for local search solvers is the presence of local minima in the search space. Local solvers can get stuck in these local solutions for extended periods, greatly damaging performance. Walksat [55], an elaboration of GSAT, adds periodic random search moves in order to escape from local minima. Local search solvers also generally are not complete, in the sense that they may search forever and not halt if no model assignment exists. This weakness is unacceptable in some domains, such as circuit verification, in which successful verification entails proving unsatisfiability. Recent work has achieved completeness for local solvers by constraining local search with learned clauses [21] and may result in increased research activity in this area.

In contrast, discrete constrained algorithms, better known as *systematic* SAT solvers, are complete, precisely because their search is systematic (i.e. does not search the same node in the search space twice). Systematic solvers operate with sets of hard constraints, usually that the clauses not be falsified by the assignment. Systematic algorithms therefore backtrack in the search space when assignments made to support clauses make the larger assignment set inconsistent. Most systematic solvers are variants of the DPLL algorithm [15, 16], pictured in Figure 1.1, which I cover in greater detail in section 1.4. DPLL is a depth-first search algorithm that builds up an assignment from the empty set, backtracking when it can no longer be extended to a solution. Systematic SAT solvers are the class of SAT algorithms directly relevant to this dissertation. Though my work deals with quantified clauses instead of propositional clauses and even solves what is technically a different problem than SAT, my solver uses the same basic structure as does DPLL, and shares many of its properties, including systematicity and completeness.

In recent years, systematic solvers have improved performance by several orders of magnitude using a variety of optimizations. Perhaps the most important of these is non-chronological backtracking integrated with conflict learning [42, 9, 66]. Both these optimizations exploit the fact that most problems contain many copies of the same subtree in their search space: in other words, solvers often encounter and solve the same satisfiability subproblem thousands of times. These optimizations detect those isomorphisms and prune them from the search space. Non-chronological backtracking analyzes the history of assignments to determine the earliest chosen assignment that contributed to a current backtracking point (backtracking points in systematic satisfiability are points in which the proposed assignment has become inconsistent), and backtracks multiple levels to that branch. This move brings the search immediately back to a node it would have eventually reached anyway, avoiding all the intervening search. Conflict learning analyzes a backtracking point and makes new clauses expressing the collective falsity of the assignments that lead to that point; these clauses allow earlier detection of this dead end. These strategies, which work well with a systematic exploration of the search space, are another important reason systematic solvers are currently dominant against local search solvers.

Other effective speedup techniques for systematic solvers include heuristics for choosing the next variable to branch upon [41, 47]; typically, these heuristics work in tandem with clause learning by trying to direct search towards learned conflicts using data about literals' frequency in recently used conflict clauses. In addition, innovations in compact representation of the clauses [47] have reduced memory requirements and increased data locality. Another material innovation has been the use of random restart [28], which addresses the problem that early literal choices can change runtime by orders of magnitude. This technique, used by a number of top solvers, restarts the search after some random period, instead of following an unlucky search to its conclusion.

Considerable research effort has also been directed at finding tractable subclasses of the SAT problem and of analyzing the boundary between tractable and intractable problems. The class of Horn formulas [20, 53] is known to be solvable in linear time, as is the class of 2-SAT formulas [3]. Other classes are known to have polynomial time, although none seem to have wide application in real-world problems. Of more practical interest are easy-hard-easy results showing that problem hardness increases dramatically as the ratio of clauses to variables approaches 4.25 [45] and then declines as quickly.

In subsequent sections I describe the way that all SAT algorithms represent and maintain state data. I will claim that this state representation is intrinsically exponential, and later present a more efficient approach.

1.4 The Core of Today's State of the Art SAT Solvers: The DPLL Algorithm

The venerable Davis-Putnam-Logemann-Loveland (DPLL) Algorithm [15,16], is a systematic and complete SAT solver. DPLL remains the core of most competitive solvers for the SAT problem and also the quantified problem formalism presented in section 1.6, FQSAT. These solvers optimize the original DPLL method using techniques such as non-chronological backtracking and clause learning. For example, Paradox, the fastest solver in a recent FQSAT testing competition (see [13] in the 2007 CADE ATP Competition [61]) is an augmentation of DPLL.

DPLL is a recursive, depth-first algorithm that builds up a set of assignments of TRUE/FALSE to problem atoms until a model is found or the assignment search space is exhausted. This set A of atom/truth-value pairs is called a *partial assignment* because typically A does not contain an assignment for every atom. I say that atoms not in $A's$ assignment are *unvalued* or have truth assignment UNVALUED. For convenience, I typically speak of A as containing positive and negative literals, e.g. I use $p, \neg q,$ as a shorthand for the corresponding atom assignments (in this case ,$\langle p, \text{TRUE}\rangle$ and $\langle q, \text{FALSE}\rangle$). In the latter case, I say that q is *falsified* by A, and also that any ground clauses containing q are falsified by A. Each time an atom assignment is added to or removed from A, the atom is said to have been *flipped* to TRUE/FALSE or back again to UNVALUED. I also speak of flipping literals, e.g. flipping atom q to FALSE might be referred to as flipping $\neg q$.

At each invocation, DPLL first checks (in line 1) if any assignments are mandated by a unit propagation inference rule. Specifically, the unit propagation rule concerns those clauses not yet true under A that have exactly one unvalued literal. In such cases we may infer that the remaining clause literal's atom must be given an assignment that will satisfy that literal (and thereby the clause). For example, if A contains $\neg p$and some clause $c = p \vee q$, then we infer A must be supplemented with q if it is ever to satisfy c (and C). Adding such assignments as soon as they may be inferred effectively prunes the lowest level of the search tree.

After unit propagation, DPLL checks (in line 2) if the current assignment has become inconsistent and backtracking is needed, and conversely, checks (in line 4) if a satisfying assignment has been found. Checking for inconsistency is straightforward: A is scanned for any two assignments that assign both TRUE and FALSE to the same atom. Checking for satisfiability entails scanning each clause c to see if some literal $l \in c$ is in A. If neither of these checks succeed, DPLL chooses some unvalued clause literal (line 6) and tries two recursive calls (lines 7-10) in which the literal or its negation is added to A; this branching recursion is called a *split* or a *branch.* Every invocation of DPLL reaching line 11 ends with FAILURE, indicating search space exhaustion.

1.5 The Subsearch Problem and the Ground Instantiation Method

In their work on an quantified satisfiability solver [26], Ginsberg and Parkes identify a central function of any satisfiability solver, a function they label *subsearch.* Subsearch is the maintenance at every solver search iteration (e.g. every call to DPLL) of crucial state data supporting the ongoing search for a logical model. As with most search algorithms, a large proportion of the code lines of satisfiability solvers concern state maintenance. For example, in Figure 1.1, lines 1, 2, 4 and 6 of DPLL all perform subsearch functions.

One of Ginsberg and Parke's contributions is the claim that all subsearch functions perform different aspects of the same fundamental calculation: the degree to which the current assignment A satisfies or falsifies the set of ground clauses implied by the

```
procedure DPLL($C$,$A$)
1: $A$ := UNIT-PROPAGATE($A$)
2: if $A$ contains inconsistent atom bindings then
3:     return FAILURE
4: if every $c \in C$ is TRUE under $A$ then
5:     return SUCCESS
6: $l$ := some clause literal not assigned a value by ; $a$:= $l$'s atom
7: for $val$ in {TRUE,FALSE} do
8:     $A' := A \cup \langle a, val \rangle$
9:     if DPLL($C$,$A'$) == SUCCESS then
10:        return SUCCESS
11: return FAILURE
```

Figure 1.1: The Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Given a SAT problem C and a partial assignment A, to compute DPLL(C,A):

problem theory. Ginsberg and Parkes also claim that for typical problems of interest, this subsearch calculation is where most of the work of the larger satisfiability algorithm resides; indeed, they prove that the subsearch function itself is NP-complete in the size of the quantified theory for FQSAT problems.

In this document, Ginsberg and Parkes' notion of subsearch is rephrased by introducing the idea of *partially falsified ground clauses* (or PFGCs for short). If we label the set of ground clauses $\mathcal{G}$ implied by the problem theory (in the case of SAT, $\mathcal{G}$ is equivalent to the set of problem clauses C), then a partially falsified ground clause c is a clause in $\mathcal{G}$ that has some number of its literals falsified by the current assignment, and the rest unvalued. Lines 1, 2, 4 and 6 in Figure 1.1 can all be viewed as gathering PFGCs of different classes, classes defined by the number of false and unvalued literals in the PFGC. Computing unit propagation (line 1) involves finding PFGCs for which every literal but one is falsified. Detecting that the current assignment is inconsistent (line 2) involves finding PFGCs for which every literal is falsified[1]. Detecting that the current assignment is in fact a model (line 4) can be achieved by verifying that every PFGC has a satisfied literal, or equivalently, that it has at least one literal that is not false or unvalued . Finally, in line 6, the set of clauses we may split upon is precisely the set of PFGCs failing the model test, because we want to avoid choosing those clauses that are already made true by the current assignment A. A last, optional subsearch function (not implemented in DPLL, but implemented as an optional feature in MACE2, a DPLL-style solver) is to find PFGCs having exactly k unvalued literals. This function can be used to implement a heuristic function for literal choice (line 8). The idea is to choose literals from unsatisfied clauses having as few unvalued literals as possible, with the hope that splitting upon these literals will cause the current assignment to either succeed or backtrack sooner rather than later.

1.5.1 The Ground Instantiation Method

Note that DPLL does not prescribe how subsearch functions are to be computed or represented. However, in practice, virtually all SAT solvers (including DPLL-style solvers) keep an explicit record for each ground clause c in the set of problem clauses C. As changes are made to the current assignment A, every clause c affected by the change is incrementally updated, often by updating tallies such as the number of clause literals in c that are falsified by A. I label this approach to subsearch the *ground instantiation* approach in order to contrast it with my own, which uses a quantified representation for the logical satisfiability problem. The next section discusses the problem with the ground instantiation approach to subsearch first noted in [26], namely that it has inherently exponential complexity and is inefficient in practice. In contrast, my work explores a new way of representing and solving satisfiability problems using quantifiers: the chief advantage of this approach is that it performs subsearch more efficiently.

Interestingly, all the SAT methods described in section 1.3 perform subsearch in

[1]In DPLL this is done indirectly, by computing unit propagations that conflict with each other

roughly the same fashion as DPLL, by tracking the degree to which a current assignment solves a set of ground clauses. Therefore, because my method primarily concerns only the underlying subsearch representation, it ought to be applicable to all SAT solving approaches, not just to DPLL.

1.5.2 The Problem with The Ground Instantiation Method

The problem with the ground instantiation method is that it uses an intrinsically exponential representation unnecessarily. Almost all SAT problems of interest (aside from randomly-generated ones, which are primarily of theoretical interest) can be more succinctly and naturally specified in first order logic (FOL). Indeed, I believe that the great majority of SAT problems of interest were first formulated as FOL, and only later translated to propositional form. This claim cannot be easily proven because repositories of SAT problems typically do not provide a key giving the original semantics of literals in a given SAT problem. However, the sheer size of many problems (some machine verification problems are over 100Mb in size) at least indicates they are machine-generated from some smaller semantic form. One source of benchmark problems that does provide descriptions is SATLIB [33]. These descriptions of problems from areas such as AI planning, circuit analysis, and mathematics, make it clear that simpler problem descriptions which could be represented in first order logic are being used to generate the sets of ground clauses used by SAT.

An example of how a quantified logical sentence might be translated to SAT will demonstrate the compactness of quantified representations. Suppose we have a planning problem whose domain involves planes flying between locations delivering shipments. A planning constraint requiring that every plane p have a location l at a given time t might be expressed logically as

$$\forall p,t\,\exists l.\,at(p,l,t) \tag{1.1}$$

Now, since SAT theories are propositional, this must be translated into clauses of the form

$$at(plane1, loc1, time1) \vee at(plane1, loc2, time1) \vee \cdots \vee at(plane1, loc3, time1) \quad (1.2)$$

which expresses the constraint for one plane and time for a set of possible locations. Typically, the original constraint is instantiated into sets of clauses using constants representing the sets of planes, locations, and times that are of interest:

$$\begin{gathered}
at(plane1, loc1, time1) \vee at(plane1, loc2, time1) \vee \cdots \vee at(plane1, loc3, time1), \\
at(plane1, loc1, time2) \vee at(plane1, loc2, time2) \vee \cdots \vee at(plane1, loc3, time2), \\
\vdots \\
at(plane1, loc1, time10) \vee at(plane1, loc2, time10) \vee \cdots \vee at(plane1, loc3, time10), \\
at(plane2, loc1, time1) \vee at(plane2, loc2, time1) \vee \cdots \vee at(plane2, loc3, time1), \\
\vdots \\
at(plane6, loc1, time10) \vee at(plane6, loc2, time10) \vee \cdots \vee at(plane6, loc3, time10)
\end{gathered} \quad (1.3)$$

Clearly, the set of resulting propositional clauses can be exponentially larger than the original FOL clause. SAT solvers often have difficulty dealing with very large problems and domains because of this phenomenon, called *clause blowup*: the number of clauses grows so great that they exhaust available memory, or at least cause significant slowdown due to cache misses [2].

Furthermore, ground instantiation algorithms also perform computations at every search node that are exponential even in the best case. I demonstrate this using a similar planning example. Suppose our planning domain has 100 planes ($jet1 \ldots jet100$), 100 airports for locations ($bos \ldots chi \ldots lax \ldots sfo$), and 100 times ($1200am, 1215am \ldots 1145pm$[3]), and that we use the following quantified sentence to model the planes' movements over time::

$$S: \forall p, t_1, t_2, l_1, l_2. (loc(p, l_1, t_1) \vee move(p, l_1, l_2, t_1, t_2)) \rightarrow loc(p, l_2, t_2) \quad (1.4)$$

[2] one possible exception is Lazy WalkSAT [57], which claims to instantiate ground clauses on demand

[3] I ask the reader not to notice that this list is really has only 96 times in it!

In the sentence, the variable p stands for a plane, t_1 and t_2 for times, and l_1 and l_2 stand for locations. The sentence says that if a plane is located at a given place at time t_1 and executes a movement action to another place starting at time t_1 and ending at time t_2, then the plane will be at the second location at time t_2. A ground instantiation solver would turn this sentence into an equivalent (implicitly) quantified clause:

$$c:\ \neg loc(p,l_1,t_1) \vee \neg move(p,l_1,l_2,t_1,t_2) \vee loc(p,l_2,t_2) \tag{1.5}$$

The ground instantiation solver would then instantiate the above clause into ground clauses using the given domain of planes, locations, and times:[4]

$$\begin{gathered}
\neg loc(jet1,bos,1215am) \vee \neg move(jet1,bos,bos,1215am,1215am) \vee loc(jet1,bos,1215am),\\
\neg loc(jet1,bos,1215am) \vee \neg move(jet1,bos,bos,1215am,1230am) \vee loc(jet1,bos,1230am),\\
\neg loc(jet1,bos,1215am) \vee \neg move(jet1,bos,bos,1215am,1245am) \vee loc(jet1,bos,1245am),\\
\vdots\\
\neg loc(jet1,bos,1215am) \vee \neg move(jet1,bos,chi,1215am,1215am) \vee loc(jet1,chi,1215am),\\
\neg loc(jet1,bos,1215am) \vee \neg move(jet1,bos,chi,1215am,1230am) \vee loc(jet1,chi,1230am),\\
\vdots\\
\neg loc(jet100,sfo,1145pm) \vee \neg move(jet100,sfo,sfo,1145pm,1145pm) \vee loc(jet100,sfo,1145pm)
\end{gathered} \tag{1.6}$$

Suppose also that we have a ground atom $a_1 = loc(jet7,lax,2pm)$, meaning that $jet7$ is at airport lax at time $2pm$. If we add $a_1 :=$ TRUE to the current assignment, then to maintain the problem state a solver using ground instantiation must update all the ground clauses that contain atom a_1 or its negation, $\neg a_1 = \neg \text{loc(jet7, lax, 2pm)}$. For the moment, consider only those clauses containing the negation. Now, one ground clause is created in equation 1.6 for each possible binding between domain members and the variables in the clause of equation 1.5; since there are five clause variables, the total number of ground clauses is 100^5. The number of these that contain negated atom $\neg a_1$, which binds the three clause variables p, l_1, and t_1, must be $100^{5-3} = 100^2$. So,

[4]Here, for simplicity, I allow instantiations in which a plane moves from one location to the same location and even moves between two locations at the same time. A clause actually used in planning would be more constrained, by, e.g. requiring that $t_1 \neq t_2$.

the solver must update 10,000 clauses when only one atom is added to the assignment. This key point may be expressed more intuitively. When a plane p begins movement at some location l_1 and time t_1, the solver is updating clauses concerning every possible location (l_2) that the plane might enter at every possible future time (t_2). All these updates occur despite the fact that a given hypothetical plane is unlikely to ever reside in the vast majority of these future time-space tuples. One would think that a subsearch computation that used only those facts that are actually TRUE would be more efficient.

More generally, at each search iteration, ground instantiation algorithms update a number of ground clauses exponential in the number of variables in the original quantified clause. In contrast, this dissertation explores a subsearch computation that avoids the exponential computation by using only literals from the current assignment, i.e. only those facts that are actually TRUE or FALSE (as opposed to UNVALUED). These literals are combined using networks of joins similar to database joins. This approach to subsearch requires that the problem be represented using quantified clauses like the one in equation 1.5. This representation in turn requires me to reformulate the satisfiability problem to use quantified clauses. In the sections to come, I first present this new problem formalization, called FQSAT. Then I discuss a well-known algorithm for solving FQSAT by translation to SAT. This algorithm, MACE2, automatically performs the kind of ground instantiation described in this section and then solves the resulting SAT problem using a DPLL-style solver. MACE2 therefore is an exemplar of solving FQSAT via translation to SAT and the use of a ground instantiation solver. My system, JOINSAT, is an exemplar of solving FQSAT using a quantified representation and join networks. Subsequent chapters qualitatively and experimentally compare the two systems and their associated methods.

1.6 FQSAT: A Formalism for Quantified Satisfiability

In this section, I formalize a quantified logical satisfiability problem, FQSAT, that is already in wide use, although not under any name that I know of.

Problem The Finite Domain Quantified Satisfiability (FQSAT) Problem.

Given:

1. a finite domain universe D consisting of d distinct elements, but whose further properties are unknown; as a convenience, these elements are referred to using the natural numbers $1 \dots d$;
2. a finite theory $\mathscr{T}$ of sentences of first order logic (FOL);
3. implicitly, a language $\mathscr{L}$ containing the predicates, constants, and function symbols in $\mathscr{T}$;

Return either:

1. a model $\mathscr{M}$ for $\mathscr{T}$, consisting of D and an interpretation function $\mathscr{I}$ expressed as:
   (a) a mapping of each predicate instance whose arguments are members of D (e.g. $p(1,1,3)$) to one of $\{\text{TRUE},\text{FALSE}\}$;
   (b) a mapping of each constant (e.g. c) and function instance in $\mathscr{T}$ whose arguments are members of D (e.g. $g(1,2)$) to some member of D;

   or:
2. FAILURE, if no such model $\mathscr{M}$ exists.

For example, given some domain universe D of cardinality 2, and a theory $\mathscr{T}$ consisting of the two sentences $\forall x. \neg p(x) \lor q(x)$ and $\exists x. p(x)$, an FQSAT solver might return a model $\mathscr{M}$ that assigns FALSE to $p(1)$ and $q(2)$ and TRUE to $p(2)$ and $q(1)$.

In contrast with the problem of satisfiability of unrestricted first order theories (presented in chapter 5, Related Work), FQSAT, which requires a finite domain, is fully decidable. This is made clear by reflecting that $\mathscr{T}$ is of finite length, as is D, so the set of possible mappings referred to in (1) must be finite as well, and therefore the set of potential models must be finite. So, an FQSAT solver that systematically explores the potential models will eventually return an answer. However, FQSAT is still intractable in the general case: the quantified boolean satisfiability (QBF) problem is PSPACE-complete [11], and QBF problems can be modeled using FQSAT by using a domain universe of {TRUE,FALSE}. So, it is likely that any algorithm for FQSAT will be require exponential time in the worst case.

Typically, an FQSAT solver is used to solve a slightly more general problem, the problem of whether a given theory $\mathscr{T}$ has a model for some domain less than a given size d_{end}. Usually the solver first tries to find a model for $d = 1$, and if that fails, $d = 2$, and so on until a model is found or d_{end} is reached.

1.7 Using SAT Methods to Solve FQSAT: the MACE2 Solver

In this section, I present MACE2 [43], an FQSAT solver that transforms problems into SAT problems and solves them using an implementation of DPLL. Other FQSAT solvers apply the same transformations and also use DPLL, and some make use of various representational and algorithmic optimizations that MACE2 lacks (e.g. clause learning by Paradox [13]). However, all of these more advanced solvers still use the same basic representation and algorithm for subsearch computation, namely, the ground instantiation of quantified clauses into ground clauses and then the incremental update of each ground clause as solving progresses.

It is certainly important for any new method to support the latest solver optimizations, and I argue that JOINSAT could do so in section 2.4.5. However, I believe that comparing JOINSAT with optimized FQSAT solvers would muddy the comparison of the core algorithmic differences between a join-based and a ground instantiation solver. Accordingly, this dissertation will focus upon the differences between JOINSAT and MACE2, and will judge JOINSAT's success by its performance relative to MACE2.

The MACE2 algorithm is shown in Figure 1.2. In the next several sections, MACE2's preprocessing (lines 1-7 of the figure) is presented in detail, so that we may see how

```
procedure MACE2($\mathscr{T}, D$)
1: $\mathscr{T}_1$ := ExistentialsToFunctions($\mathscr{T}$)
2: $\mathscr{T}_2$ := Flatten($\mathscr{T}_1$)
3: $C_1$ := FunctionConstraints($\mathscr{T}_1, D$)
4: $C_2$ := SymmetryConstraints($\mathscr{T}_1, D$)
5: $QC_1$ := QuantifiedCNF($\mathscr{T}_2$)
6: $QC_2$ := ClauseSplit($QC_1$)
7: $C$ := Instantiate($QC_2, D$) $\cup C_1 \cup C_2$
8: $S$ := MACE2-Initialize($C$)
9: return MACE-DPLL($S$)
```

Figure 1.2: The MACE2 algorithm. Given an FQSAT problem $(\mathscr{T}, D)$, to compute MACE2$(\mathscr{T}, D)$:

MACE2 translates an FQSAT problem into the set of ground clauses required by a SAT solver.

1.7.1 MACE2's Use of Domain-Grounded Clauses

Before proceeding further with an account of MACE2, I explain my use of "ground" clauses whose arguments are members of the domain universe (e.g. $p(1,2)$), a usage which may seem not well-founded or simply confusing. I use natural numbers in place of constant symbols as a shorthand, a syntax that MACE2 also accepts. Using this shorthand makes examples much more concise, but first its meaning must be made clear. Every problem theory $\mathscr{T}$ written with this syntax is a shorthand for a well-formed FOL theory $\mathscr{T}'$ with the same satisfiability properties, i.e. $\mathscr{T}$ is satisfiabile iff $\mathscr{T}'$ is satisfiable. Theory $\mathscr{T}'$ is formed as follows. For each natural number k used in the problem theory $\mathscr{T}$, replace all appearances of k in $\mathscr{T}$ with a new constant symbol c, so e.g. $p(1,1)$ becomes $p(c,c)$, and $p(1,2)$ becomes $p(c,d)$. An equality constraint is also added for each new constant, e.g. $=(c,c)$, and an inequality constraint expressed as a predicate for each pair of new constants, e.g. $\neq (c,d)$.

Now, MACE2 treats numbers in theories as special constants whose interpretation is bound to the appropriate domain element, e.g. we may think of $p(1,1)$ as a stand-in for $p(c,c)$, but with the added requirement that any satisfying model for $\mathscr{T}$ must interpret the constant c as 1. So, we may think of a numerical theory $\mathscr{T}$ as a well-formed FOL theory $\mathscr{T}'$ given to MACE2 but accompanied with a set of restrictions upon the models to be explored. MACE2 will always flip new literals according to these restrictions, e.g. when trying to satisfy the literal $p(1,1)$ it will add $\langle p(1,1), \text{TRUE}\rangle$, not $\langle p(2,2), \text{TRUE}\rangle$ or $\langle p(c,c), \text{TRUE}\rangle$.

The foregoing shows that if MACE2 finds a model for $\mathscr{T}$, then that model also satisfies $\mathscr{T}'$, because $\mathscr{T}$ is really $\mathscr{T}'$ plus restrictions on models to explore. Now I show the converse, that $\mathscr{T}'$ is satisfiable implies $\mathscr{T}$ is satisfiable. This is a simple argument of isomorphism. Suppose $\mathscr{T}$ contains the numbers $1 \dots n$, and $\mathscr{T}'$ replaces instances of these numbers with constants $c_1 \dots c_n$. Now, our model $\mathscr{M}$ must contain an interpretation of these constants: suppose they are interpreted as domain members $d_1 \dots d_n$. These domain members must all be distinct from one another to satisfy the inequality constraints. Then there must exist some model $\mathscr{M}'$, isomorphic to $\mathscr{M}$, in which $c_1 \dots c_n$ are interpreted as $1 \dots n$ instead of $d_1 \dots d_n$.

1.7.2 MACE2 Preprocessing: Converting FQSAT Problems to Clauses

For efficiency, most DPLL-style FQSAT solvers represent logical theories in conjunctive normal form (CNF) as sets of disjunctive clauses. These clauses may be either ground clauses (e.g. $(\neg a \vee b \vee \neg c)$) or *quantified clauses* in which the logical variables are implicitly universally quantified, e.g. $p(x) \vee \neg q(x,y) \vee \neg r(y,z)$. Regardless of the clause type the solver works with, it must be able to input the full syntactic range of FOL theories and convert them to sets of clauses. In this section, I detail the processes by which MACE2 converts FOL theories to ground clauses. Some of these processes are reused by JOINSAT, which operates on quantified clauses.

1.7.2.1 Converting Existential Variables to Skolem Functions

The first step in converting to clause form is to eliminate any existential variables from sentences of the theory; in MACE2, this is performed by the procedure EXISTENTIALSTOFUNCTIONS. For a given sentence, EXISTENTIALSTOFUNCTIONS replaces each occurrence of an existentially quantified variable x with a Skolem function f_x. The arguments to f_x consist of those universally quantified sentence variables having x in their scope. For example, in the sentence

$$s : \forall v \exists w \forall x \exists y \forall z. \neg((\neg p(x) \wedge q(v,y)) \vee (r(w,g(h(x))) \wedge \neg s(z,x))) \tag{1.7}$$

the existentially quantified variable w is within the scope of the universally quantified variable v, so the Skolem function defined for w is $f_w(v)$. In this case, the one occurrence of w is replaced by f_w:

$$s' : \forall v,x \exists y \forall z. \neg((\neg p(x) \wedge q(v,y)) \vee (r(f_w(v),g(h(x))) \wedge \neg s(z,x))) \tag{1.8}$$

The remaining existentially quantified variable y is then replaced with a Skolem function $f_y(v,x)$:

$$s'' : \forall v,x,z. \neg((\neg p(x) \wedge q(v,f_y(v,x))) \vee (r(f_w(v),g(h(x))) \wedge \neg s(z,x))) \tag{1.9}$$

1.7.2.2 Flattening: Converting Functions into Predicates

Now the working theory contains instances of functions, some skolem, some not. The next step is to eliminate all these functions (using procedure FLATTEN, shown in algorithm), replacing them with functional predicates. This step is referred to as "flattening" because at its conclusion each sentence contains only flat literals whose arguments are logical variables. For each n-ary function symbol (e.g. $f(x,y)$) found in the theory, an $n+1$-ary predicate $p_f(x,y,z)$ is defined that has the semantics $p_f(x,y,z) \iff f(x,y) = z$. Each instance of f in the theory is replaced as follows. If some literal l contains an

procedure FLATTEN($\mathscr{T}$)

```
1: while $l$ := some literal in some sentence $s$ of $\mathscr{T}$ that has an instance $f(\overrightarrow{t})$, where
   $f$ is a function and $\overrightarrow{t}$ is a vector of terms do
2:     $l'$ := $l$ with $f(\overrightarrow{t})$ replaced by a new variable $y$ not already in $s$
3:     $l''$ := $\neg p_f(\overrightarrow{t}, y)$
4:     replace $l$ in $s$ with $l'' \vee l'$
5: return $T$
```

Figure 1.3: FLATTEN. Given a universally quantified theory $\mathscr{T}$, to compute FLATTEN($\mathscr{T}$):

instance $f(t_1,t_2)$ of f, l is replaced with the disjunction $\neg p_f(t_1,t_2,z) \vee l'$, where l' is l with z substituted for $f(t_1,t_2)$. In this scheme, the new variable z is implicitly universally quantifed. The replacement preserves most of the original semantics, in the sense that the sentence is only required to be satisfied for those values of z such that $f(t_1,t_2) = z$. The remaining semantics of a functional occurrence, namely the onto and one to one properties, are restored in the next section.

Let us work through how FLATTEN would eliminate some of the functions in the output sentence of the prior section, s'':

$$s'' : \forall v,x,z.\neg((\neg p(x) \wedge q(v, f_y(v,x))) \vee (r(f_w(v), g(h(x))) \wedge \neg s(z,x))) \tag{1.10}$$

Suppose that FLATTEN first eliminates the functional occurrence $h(x)$. FLATTEN first creates a new variable i not found in s'' and replaces $h(x)$ with i in the literal $r(f_w(v), g(h(x)))$:

$$l' : r(f_w(v), g(i)) \tag{1.11}$$

FLATTEN then makes the appropriate instance of the functional predicate p_h, with the new variable i as the final argument:

$$l'' : \neg p_h(x,i) \tag{1.12}$$

Finally, the original literal $r(f_w(v), g(h(x)))$ is replaced in s with $l'' \vee l'$:

$$s'' : \forall v \forall x \forall z. \neg((\neg p(x) \wedge q(v, f_y(v,x))) \vee ((\neg p_h(x,i) \vee r(f_w(v), g(i))) \wedge \neg s(z,x))) \tag{1.13}$$

The same procedure is used to eliminate $g(i)$, obtaining:

$$s'' : \forall v \forall x \forall z. \neg((\neg p(x) \wedge q(v, f_y(v,x))) \vee ((\neg p_h(x,i) \vee \neg p_g(j) \vee r(f_w(v), j)) \wedge \neg s(z,x))) \tag{1.14}$$

The rest of the functional occurrences may be removed similarly.

1.7.2.3 Flattening: Adding Functional Constraints

After functional instances are replaced with corresponding functional predicates, we must still ensure that these predicates have all the normal semantics of functions, including the one-to-one and onto properties. MACE2 models these by adding additional ground clauses to the theory. First, clauses are added to enforce the one-to-one property, which says that for a given set of arguments, a function can have only one output. These semantics are expressed for a function f by adding ground clauses of the form $\neg p_f(\vec{d}, d_1) \vee \neg p_f(\vec{d}, d_2)$, where p_f is the appropriate functional predicate, $\overrightarrow{d}$ is a set of domain value arguments of f , and d_1, d_2 are two distinct domain values. For clarity in this context, I label the domain size d_{MAX}. These clauses are added in line 6 of FUNCTIONCONSTRAINTS. For example, given a function $f(x,y)$ and argument bindings $x = 1, y = 1$, the first clause added will be $\neg p_f(1,1,1) \vee \neg p_f(1,1,2)$, expressing that either $f(1,1) = 1$ is false or $f(1,1) = 2$ is false.

Next, clauses are added (in line 9 of FUNCTIONCONSTRAINTS) to enforce the onto property, which mandates that for a given set of arguments, a function must have at least one output. These clauses have the form

```
procedure FunctionConstraints($\mathscr{T}, D$)
1: $C := \phi$
2: for all functions $f(\vec{x})$ of arity $k$ found in $\mathscr{T}$ do
3:     for all vectors $\overrightarrow{d}$ of arity $k$ of $d \in D$ do
4:         for all $i \in 1 \ldots d_{MAX}$ do
5:             for all $j \in i+1 \ldots d_{MAX}$ do
6:                 add the ground clause $\neg p_f(\vec{d}, i) \vee \neg p_f(\vec{d}, j)$ to $C$
7:         add the ground clause $p_f(\vec{d}, 1) \vee p_f(\vec{d}, 2) \vee \cdots \vee p_f(\vec{d}, d_{MAX})$ to $C$
8: return $C$
```

Figure 1.4: FunctionConstraints. Given an FQSAT problem $(\mathscr{T}, D)$, to compute FunctionConstraints($\mathscr{T}, D$):

$$p_f(\vec{a}, 1) \vee p_f(\vec{a}, 2) \vee \cdots \vee p_f(\vec{a}, d_{MAX}) \tag{1.15}$$

where $\overrightarrow{a}$ is some set of domain values and d_{MAX} is the domain size, and they express that $f(\overrightarrow{a})$ is equal to some domain element in D. For example, given function $f(x,y)$, the first onto clause to be added will be

$$p_f(1,1,1) \vee p_f(1,1,2) \vee \cdots \vee p_f(1,1,d_{MAX}) \tag{1.16}$$

expressing that $f(1,1)$ is equal to some domain element in D.

1.7.2.4 Adding Constraints to Break Symmetries

MACE2 also adds constraints to eliminate some kinds of isomorphic models; this process is also known as *symmetry breaking*. Two models are symmetric in this context if their interpretation functions can be made identical by swapping all instances of two domain members. For instance, if interpretation $\mathscr{I}_1$ maps function $f(x)$ to 2, $p(1)$ to TRUE, and $p(2)$ to FALSE, while interpretation $\mathscr{I}_2$ maps function $f(x)$ to 1, $p(2)$ to TRUE, and $p(1)$ to FALSE, then the corresponding models are symmetric. FQSAT solvers constrain the search space so that only one of a set of isomorphic models are found; because the remaining model can still be found, this does not compromise completeness.

The search space is constrained by adding propositional clauses specifying allowable return values for problem functions, or in MACE2's case, problem constants only. A constant c_1 is picked arbitrarily and constrained to return domain element 1 (this is expressed a using functional predicate p_{c_1}). The next arbitrary constant is constrained to return either element 1 or element 2, and so on:

$$\begin{gathered} p_{c_1} = 1 \\ p_{c_2} = 1 \vee p_{c_2} = 2 \\ p_{c_3} = 1 \vee p_{c_3} = 2 \vee p_{c_3} = 3 \\ \vdots \\ p_{c_d} = 1 \vee p_{c_d} = 2 \vee \cdots \vee p_{c_d} = d \\ p_{c_{d+1}} = 1 \vee p_{c_{d+1}} = 2 \vee \cdots \vee p_{c_{d+1}} = d \end{gathered} \tag{1.17}$$

These propositional clauses are created in SYMMETRYCONSTRAINTS (line 4 of Figure 1.2).

1.7.2.5 Conversion to Clause Form

At this stage, apart from the ground clauses specified in the last two sections, the working theory consists of universally quantified logical sentences whose nesting of conjunctions, disjunctions and negations may be arbitrarily deep, e.g.

$$\forall x,y,z.((-p(x) \vee \neg q(x,y)) \rightarrow P(y)) \vee (\neg r(z) \rightarrow q(x,z)) \tag{1.18}$$

We may now drop the quantifiers and assume every variable is implicitly universally quantified. However, most DPLL-style solvers (but not all: see [62]) use a CNF representation because its simplicity makes processing more efficient. Therefore, these complex sentences still need to be converted to CNF form, i.e. to a set of disjunctive clauses. While this may be accomplished using a series of classical rule translations, the resulting set of clauses may have size exponentially larger than the original theory [48].

Therefore, a special translation process first formulated by Tseitin [63] and later optimized by others [48, 17] is used.

The essential idea of this process is to first convert a sentence into a tree structure based on its connectives, and then disconnect the branches of the tree from the bottom up using pairings of new literals and new equivalence clauses. For example, to eliminate the implication $(\neg r(z) \rightarrow q(x,z))$, a new literal $B(x,z)$ and an equivalence $B(x,z) \equiv (\neg r(z) \rightarrow q(x,z))$ would be introduced. The equivalence translates (using classical logic rules) into the set of clauses $\neg B(x,z) \vee r(z) \vee q(x,z)$, $\neg r(z) \vee B(x,z)$, and $\neg q(x,z) \vee \neg B(x,z)$. Therefore, the sentence in equation 1.18 is equivalent to the set of sentences/clauses

$$\begin{gathered}((-p(x) \vee \neg q(x,y)) \rightarrow P(y)) \vee B(x,z) \\ \neg B(x,z) \vee r(z) \vee q(x,z) \\ \neg r(z) \vee B(x,z) \\ \neg q(x,z) \vee \neg B(x,z)\end{gathered} \tag{1.19}$$

This translation process continues until the top sentence has been completely clausified. This produces a set of output clauses linear in the size of the original theory. Of course, since the theoretical complexity of FQSAT is at least PSPACE-complete, even a linear increase in theory size may require exponentially greater solution time. But in practice the slowdown is much smaller, perhaps a constant factor.

1.7.2.6 Clause Instantiation

The working theory still contains quantified disjunctive clauses (as well as ground clauses), but DPLL requires all its input clauses to be ground. Therefore, each quantified clause is instantiated into a set of ground clauses, which taken together express the same semantics. This occurs in the procedure INSTANTIATE (line 7 of MACE2); I defer discussion of the prior line, CLAUSESPLIT, until the next section. To explain instantiation, I introduce the notion of a *bindset*, or set of bindings between variables and domain values. The ground clauses are constructed by iterating through all the possible bindsets for the set of clause variables and applying each bindset to the quantified clause. For example, given variables x and y and a domain size $d = 2$, the set of possible bindsets is:

$$B: \{\{x=1, y=1\}, \{x=1, y=2\}, \{x=2, y=1\}, \{x=2, y=2\}\} \tag{1.20}$$

Given the clause

$$\neg p(x) \vee q(x,y) \vee \neg r(y), \tag{1.21}$$

the corresponding set of four ground clauses, containing one ground clause for each bindset $\beta \in B$, is constructed by applying each bindset to the quantified clause. The resulting clauses are as follows:

$$\begin{aligned} \{ \quad & \neg p(1) \vee q(1,1) \vee \neg r(1), \\ & \neg p(1) \vee q(1,2) \vee \neg r(2), \\ & \neg p(2) \vee q(2,1) \vee \neg r(1), \\ & \neg p(2) \vee q(2,2) \vee \neg r(2) \quad \} \end{aligned} \tag{1.22}$$

We see above that the INSTANTIATE procedure creates a number of clauses exponential in the number of clause variables. As discussed in section 1.5.2, this can lead to significant memory blowup, so that MACE2 is not feasible for very large problems. This is mitigated somewhat by a postprocessing step in which ground clauses subsumed by another clause (e.g. as $b \vee \neg c \vee d$ is subsumed by $b \vee \neg c$) are pruned away. This can amount to a considerable savings when combined with unit resolution, in which any unit clauses in the theory are resolved against other clauses (e.g. the unit clause $\neg e$ can resolve with $b \vee \neg c \vee d \vee e$, producing the clause $b \vee \neg c \vee d$, which then subsumes $b \vee \neg c \vee d \vee f$).

1.7.2.7 Clause Splitting: A Powerful Optimization

Now that the mechanism and potential complexity of instantiating clauses is clear, I sketch an optimization in wide use: clause splitting (shown as CLAUSESPLIT in MACE2 line 6). This recent optimization for MACE2-style ground solvers [49,13] improves their performance by an order of magnitude or more. The idea of clause splitting is to rewrite the quantified clauses so that they have fewer variables. As shown in the prior section, the number of ground clauses created by INSTANTIATE is exponential in the number of clause variables, so there is considerable opportunity for optimization by rewriting. The rewriting process introduces a new predicate C and splits a clause into two clauses supplemented by instances of the new predicate. For example, given a complex clause

$$c : \neg p(u,v) \vee q(v,w) \vee \neg r(w,x,y) \vee s(y,z) \tag{1.23}$$

one possible split of he original clause c into two new clauses c_1 and c_2 is the following:

$$\begin{aligned} c_1 &: \neg p(u,v) \vee q(v,w) \\ c_2 &: \neg r(w,x,y) \vee s(y,z) \end{aligned} \tag{1.24}$$

A new literal $C(w)$ is then introduced that contains all variables the two clauses have in common. Positive and negative instances of $C(w)$ are appended to c_1 and c_2, respectively:

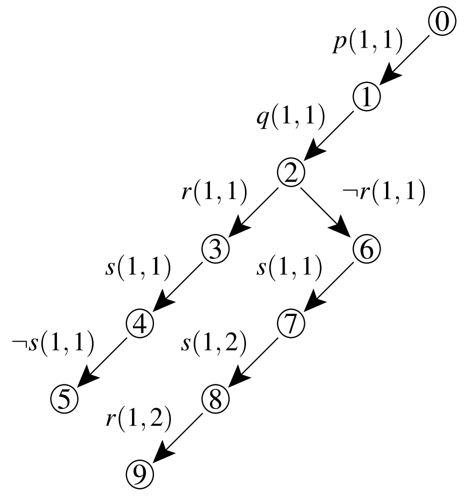
$$\begin{aligned} c_1' &: \neg p(u,v) \vee q(v,w) \vee C(w) \\ c_2' &: \neg r(w,x,y) \vee s(y,z) \vee \neg C(w) \end{aligned} \tag{1.25}$$

The resulting pair of clauses is now equivalent to c and will require many fewer instantiations because they possess fewer clause variables. If we suppose a domain size of 5, then clause c will require $5^6 = 15,625$ instantiations, while c_1' will require $5^3 = 125$ instantiations, and c_2' will require $5^4 = 625$ instantiations.

Recent work in clause instantiation [13] involves efficiently selecting the most advantageous split (a clause with n literals has 2^n possible splits).

$$
\begin{aligned}
C: \quad \{ \quad & \neg p(v,w) \vee \neg q(w,x) \vee r(x,y) \vee s(y,z), \\
& p(1,1), \\
& q(1,1), \\
& \neg r(1,1) \vee s(1,1), \\
& \neg r(1,1) \vee \neg s(1,1) \; \}
\end{aligned}
$$

(a) An example set of quantified clauses



(b) A search space history for MACE2-DPLL's run on the example set C

Figure 1.5: A sample set of clauses and MACE-DPLL's search history.

1.7.3 MACE2's Implementation of DPLL

I now examine in depth MACE2-DPLL, MACE2's implementation of DPLL, depicted in Figure 1.7. For this purpose, I introduce an example set C of quantified clauses, shown in Figure 1.5; the figure also shows the search space MACE2 will traverse in trying to satisfy the clauses. First, MACE2-INITIALIZE is called, setting up the data structures needed for the initial state; these structures are all included under an umbrella structure S , pictured in Figure 1.6.

S contains two tables: a table *atoms* containing a record for each ground atom, and a table *clauses* containing a record for each ground clause. Each entry *atom* in *S.atoms* has a field *value*, represented as an (initially empty) list of the truth values assigned to the atom by the current assignment A. A is therefore represented implicitly, as simply the set of atoms in *S.atoms* that have a non-empty *value*. Each atom also contains lists of the ground clauses in which it appears in negated and non-negated form: these lists are $atom.clauses[\text{FALSE}]$ and $atom.clauses[\text{TRUE}]$, respectively. In this context, each ground clause is represented as an integer that can be used as an index into *S.clauses*, described next. For example, in the figure, atom $p(1,1)$ is the first entry in *S.atoms*, and its list of $clauses[\text{FALSE}]$ contains clauses $c_1 \ldots c_8$, which are the first eight entries in *S.clauses*.

Each entry in *S.clauses* contains a list *lits* of literals contained in that ground clause; the literals are here represented as integers (positive for non-negated literals, negative for negated literals). The absolute value of a literal is also the index into *S.atoms* used for access to the corresponding atom, e.g. literal $\neg p(1,1)$ is represented as integer -1, because atom $p(1,1)$ is the first entry in *S.atoms*. Each *S.clauses* entry also contains various tallies for that clause, for example, *num_lits_sat*, which stores the number of clause literals that are satisfied by the current assignment A. The other tallies are *num_neglits_unvalued*, the number of negated clause literals not yet valued by the assignment, and *num_poslits_unvalued*, the number of non-negated clause literals not yet valued by the assignment. The tally *num_lits*, the total number of literals in the clause, is not actually needed by the algorithm, but is shown here for clarity.

Recall from section 1.5 that the most costly component of a DPLL-style solver, both in space and time, is the component that maintains the subsearch computation. Like all ground instantiation solvers, MACE2's approach to both representation (i.e. space) and computation (time) is exponential; we see the former already in Figure 1.6, in which exponential numbers of records representing atoms and clauses are explicitly stored. MACE2 also performs the subsearch computation in a brute-force, exponential fashion, by updating the aforementioned tallies for each clause affected by a given atom flip. At the cost of an exponential number of clause updates per atom flip, this machinery en-

S

atoms

ATOM	*value*	*clauses*	
		FALSE	TRUE
$a_1 : p(1,1)$	UNVALUED	$1 \ldots 8$	33
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a_4 : p(2,2)$	UNVALUED	$25 \ldots 32$	ϕ
$a_5 : q(1,1)$	UNVALUED	$1 \ldots 4, 17 \ldots 20$	34
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a_8 : q(2,2)$	UNVALUED	$13 \ldots 16, 29 \ldots 32$	ϕ
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a_{16} : s(2,2)$	UNVALUED	ϕ	$4,8,12,16,20,24,28,32$

clauses

CLAUSE	*num_lits*	*num_lits_sat*	*num_neglits_unvalued*	*num_poslits_unvalued*	*lits*
$c_1 : \neg p(1,1) \vee \neg q(1,1) \vee r(1,1) \vee s(1,1)$	4	0	2	2	-1,-5,9,13
$c_2 : \neg p(1,1) \vee \neg q(1,1) \vee r(1,1) \vee s(1,2)$	4	0	2	2	-1,-5,9,14
$c_3 : \neg p(1,1) \vee \neg q(1,1) \vee r(1,2) \vee s(2,1)$	4	0	2	2	-1,-5,10,15
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$c_8 : \neg p(1,1) \vee \neg q(1,2) \vee r(2,2) \vee s(2,2)$	4	0	2	2	-1,-6,12,16
$c_9 : \neg p(1,2) \vee \neg q(2,1) \vee r(1,1) \vee s(1,1)$	4	0	2	2	-2,-7,9,13
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$c_{32} : \neg p(2,2) \vee \neg q(2,2) \vee r(2,2) \vee s(2,2)$	4	0	2	2	-4,-8,12,16
$c_{33} : p(1,1)$	1	0	0	1	1
$c_{34} : q(1,1)$	1	0	0	1	5
$c_{35} : \neg r(1,1) \vee s(1,1)$	2	0	1	1	-9,13
$c_{36} : \neg r(1,1) \vee \neg s(1,1)$	2	0	2	0	-9,-13

Figure 1.6: State Data Structure S created by MACE2-Initialize.

ables MACE2 to determine, for any ground clause, whether it is currently TRUE, FALSE, or UNVALUED under the current assignment, and allows MACE2 to select literals for propagation and atoms for splitting. Despite the theoretical complexity, this scheme has the virtue of simplicity: its computations consist primarily of reading lists and updating arrays, requiring few dynamic memory operations.

I represent MACE2-DPLL as using a different state structure *S* at each node in its search space, i.e. for each recursive call MACE2-DPLL copies the current state structure and uses the copy for the call. This is a presentational simplification: in the actual algorithm the original structure *S* is never copied, but is instead incrementally updated. Presenting the algorithm in this way allows us to avoid depicting backtracking (which simply undoes the work of those procedures that are of real interest) in any detail. Instead, I depict backtracking as simply throwing away the S' created for the recursive call and returning to its predecessor state *S*. Despite this simplification, the operations that I do depict faithfully present the incremental manner in which MACE2 updates the search state.

I will use Figure 1.5 to trace the search space of MACE-DPLL's run; the run begins at the root, the node labeled ⓪ in the figure. MACE-DPLL's procedures mirror those of the general DPLL algorithm, so it begins in line 1 of Figure 1.7 with unit propagation; MACE2-UNIT-PROPAGATE is depicted in detail in Figure 1.8.

MACE2-UNIT-PROPAGATE loops, beginning in its second line, until either the current assignment has become inconsistent or no more literals are found to propagate. The former condition is tested in line 2 by checking all entries in *S.atoms* to verify that no entry has a *value* of both TRUE and FALSE. The body of the loop gathers more propagation literals (lines 4-10) and then flips each one (lines 11-16). Literals are gathered by examining clause tallies: clauses with no satisfied literals and exactly one unvalued literal are candidates for propagation, because the one remaining literal must be made true for the current assignment to ever satisfy the clause. Clauses that qualify are searched for their one unvalued literal, which is pushed onto *unit_stack*; the *continue* variable is also set, to perpetuate the outer loop. In the flipping sequence, literals are flipped to

```
procedure MACE2-DPLL($S$)
 1: MACE2-UNIT-PROPAGATE($S$)
 2: if $S.atoms$ contains an atom entry whose $value$ contains both TRUE and FALSE then
 3:     return FAILURE
 4: if for every $c \in S.clauses, c.num\_neglits\_unvalued > 0$ or $c.num\_lits\_sat > 0$ then
 5:     return SUCCESS
 6: $lit$ := MACE2-SELECT-NONNEGATED-LITERAL($S$)
 7: for all $val \in \{$TRUE,FALSE$\}$ do
 8:     $S$' := copy $S$
 9:     MACE2-FLIP($S'$, absolute_value($lit$), $val$)
10:     if MACE2-DPLL($S'$) == SUCCESS then
11:         return SUCCESS
12: return FAILURE
```

Figure 1.7: MACE2-DPLL.

the value that causes their clause to be satisfied. In the initial state ⓪ of the example problem, two clauses, clause $c_{33} : p(1,1)$, and clause $c_{34} : q(1,1)$, each contain a single unvalued literal and so are eligible for propagation. These literals are each flipped to TRUE, moving the search state to node ②.

At this stage, let us look more closely at the MACE2-FLIP procedure. This procedure is where most of the work resides in MACE2, for the various clause tallies which support subsearch functions are updated here when atoms are flipped. The procedure is passed the state variable S, an integer *atom*, and a *val* of either TRUE or FALSE. In line 1, the procedure accesses the proper entry for the flipped atom in *S.atoms* (using *atom* as the key) and updates its truth *value* to *val*. The rest of the procedure loops through precomputed lists of clauses affected by the current flip, updating various tallies for them. The first list (line 2) is of those clauses containing a negated instance of the atom. Each of these clauses now has one less unvalued negative literal, so *num_neglits_unvalued* is decremented. If $val =$ FALSE, each clause now has one more satisfied literal, so *num_lit_sat* is incremented. The second list is the converse of the first list for clauses containing a non-negated instance of the atom.

In the example, when $p(1,1)$ is flipped to TRUE, *num_neglits_unvalued* is decremented for clauses $c_1 \ldots c_8$, which have negated instances of $p(1,1)$ as literals. In contrast, clause c_{33}, which contains a non-negated instance of $p(1,1)$, has both *num_poslits_unvalued* decremented and *num_lits_sat* incremented.

We now return to MACE-DPLL, which has just concluded unit propagation (line 1). MACE-DPLL next checks for an inconsistent assignment (line 2) using the same check found in MACE2-UNIT-PROPAGATE, and seeing no inconsistency at this stage, then checks if the assignment is a model (line 4). The model check reflects what I will call a *negative bias* on the part of MACE2. Like most DPLL-style solvers, MACE2 assumes that for most problems, the models that satisfy them will be mostly negative, i.e. the ratio of atoms assigned FALSE to atoms assigned TRUE will be large. Under these conditions, the following model check is advantageous. If an assignment A is ever reached such that, for each problem clause c, either (1) A explicitly satisfies c by satisfying one of

```
procedure MACE2-UNIT-PROPAGATE($S$)
 1: $continue$ := TRUE
 2: while $S.atoms$ contains no atom entry whose $value$ contains both TRUE and FALSE,
    and $continue$ = TRUE do
 3:   $continue$ := FALSE
 4:   for all $clause \in S.clauses$ do
 5:     if ($clause.num\_lits\_sat$ == 0 and $clause.num\_neglits\_unvalued$ +
        $clause.num\_poslits\_unvalued = 1$) then
 6:       for all $lit \in clause.lits$ do
 7:         $atom$ := absolute-value($lit$)
 8:         if $S.atoms[atom].value$ is empty then
 9:           push($lit$,$unit\_stack$)
10:           $continue$ := TRUE
11:   while $lit$ := pop($unit\_stack$) do
12:     $atom$ := absolute-value($lit$)
13:     if ($lit > 0$) then
14:       MACE2-FLIP($S$,$atom$,TRUE)
15:     else
16:       MACE2-FLIP($S$,$atom$,FALSE)
```

Figure 1.8: MACE2-UNIT-PROPAGATE(S).

```
procedure MACE2-FLIP($S, atom, val$)
1: push($S.atoms[atom].value, val$)
2: for all $clause \in S.atoms[atom][\text{FALSE}]$ do
3:     $clause.num\_neglits\_unvalued$--
4:     if $val = \text{FALSE}$ then
5:         $clause.num\_lits\_sat$++
6: for all $clause \in S.atoms[atom][\text{TRUE}]$ do
7:     $clause.num\_poslits\_unvalued$--
8:     if $val = \text{TRUE}$ then
9:         $clause.num\_lits\_sat$++
```

Figure 1.9: MACE2-FLIP($S, atom, val$).

c's literals; or (2) c has some negative literal unvalued by A, then the algorithm has succeeded at finding a model. This is because the current assignment A may be extended by assigning FALSE to every remaining unvalued atom, thereby satisfying all clauses of the second type. This check is performed in line 4 for each clause in *S.clauses*, using tallies maintained for this purpose. In search space node ⑤ of the example, this model check fails because, for example, clause $c_1 : \neg p(1,1) \vee \neg q(1,1) \vee r(1,1) \vee s(1,1)$ has all its negative literals falsified and no positive literals valued.

MACE2-DPLL next selects a literal for splitting (line 6), calling MACE2-SELECT-NONNEGATED-LITERAL, shown in Figure 1.10. This procedure selects a literal using the minimum unvalued literals heuristic mentioned in section 1.5. In line 1, the variable *num_poslits_unvalued* is initialized to MAXINT, the maximum representable integer value. This variable helps track the best clause seen so far by storing its number of unvalued positive literals. In lines 2-4, the procedure scans exactly those clauses that failed the model check from MACE2-DPLL line 4. From these clauses, the code isolates the clause with the minimum number of unvalued positive literals, and selects one of its unvalued positive literals (lines 4-9); the literal is returned in line 10.

procedure MACE2-SELECT-NONNEGATED-LITERAL(S)

```
1: $num\_poslits\_unvalued$ := MAXINT
2: **for all** $clause \in S.clauses$ **do**
3:   **if** ($clause.num\_lits\_sat == 0$ and $clause.num\_neglits\_unvalued = 0$) **then**
4:     **if** $clause.num\_poslits\_unvalued < num\_poslits\_unvalued$ **then**
5:       **for all** ($lit \in clause.lits$) **do**
6:         $atom$ = absolute-value($lit$)
7:         **if** ($lit > 0$ and $S.atoms[atom].value = \phi$ **then**
8:           $return\_lit$ := $lit$
9: return $return\_lit$
```

Figure 1.10: MACE2-SELECT-NONNEGATED-LITERAL(S).

This literal will be split upon in lines 7-10 of MACE2-DPLL. In the example, the clause c_1 mentioned above as failing the model test is found and its first unvalued positive literal, $r(1,1)$ is returned. MACE2-DPLL creates a branch in the search space, first flipping $r(1,1)$ to TRUE (search space node ③), then backtracking and flipping $r(1,1)$ to FALSE (search space node ⑥) if the first branch fails. In each case, as stated before, the recursive call is made using a copy S' of the current state S.

The recursive invocation of MACE2-DPLL performs unit propagation, this time using the clauses $c_{35} : \neg r(1,1) \vee s(1,1)$ and $c_{36} : \neg r(1,1) \vee \neg s(1,1)$, because $r(1,1)$ is TRUE under the current assignment. As a result, $s(1,1)$ is assigned TRUE and then FALSE in succession (search space nodes ④ and ⑤), causing the assignment inconsistency check to fail in line 2 and the second invocation of MACE2-DPLL to backtrack, returning FALSE. The original invocation of MACE2-DPLL throws away S', resuming with S at search space node ②, and begins the loop in line 9 a second time. When $r(1,1)$ is flipped to FALSE in search space node ⑥, the recursive call to MACE2-DPLL makes more progress. Clause c_1 is now eligible for propagation because its first three literals are falsified: $s(1,1)$ is accordingly flipped to TRUE (search node ⑦). A similar clause prompts the propagation of $s(1,2)$ (search space node ⑧). Now the only clauses left unsatisfied are of the form $\neg p(1,1) \vee \neg q(1,1) \vee r(1,2) \vee s(2,z)$. In MACE2-SELECT-LITERAL, the literal $r(1,2)$ is selected from one of these clauses and flipped to TRUE in yet another recursive call of MACE2-DPLL (search node ⑨). At this point, all clauses are either satisfied or have an unvalued negative literal, so MACE2-DPLL's model check returns SUCCESS.

This completes the presentation of MACE2, a state of the art solver for FQSAT problems. This detailed examination should help the reader of the following sections to understand what is fundamentally novel about my own approach.

Chapter 2

The Main Idea - Subsearch Using Join Networks

2.1 Production Systems: An Inspiration for a New Method

AI Production Systems [22,39,52] are a reasoning formalism used in research in general cognitive models and learning; the study of production systems was a thriving artificial intelligence research area in the 1980's and 1990's. Production Systems are also one of the core technologies used in the research and commercial field of Expert Systems [25]. A production system consists of a set of *production rules*, or *productions* for short, each consisting of a set of preconditions and one conclusion. The preconditions and conclusion contain predicates and variables and look something like quantified logical literals. A production system works over a series of cycles, maintaining at every cycle a *working memory* consisting of a set of active *tuples*. The sense of the term tuple follows that of databases, meaning a class label combined with an associative array of key-value pairs, the whole tuple representing some entity in the domain (e.g. a person, or a development team). Figure 2.1 shows a simple production system consisting of one production rule and a working memory of nine tuples. In the figure, keys are prefixed with "^" (e.g. ^NAME), while variables appear within brackets (e.g. <N1>).

At each cycle, for every production, the system attempts to match active tuples to rule preconditions so that (1) every precondition has a match and (2) the resulting bindings of variables in the production are consistent. Each such successful match between a rule and a set of active tuples causes the rule to "fire", adding an instance of the conclusion made using the match bindings to the set of active tuples for the next cycle. For example, using the system in Figure 2.1, tuples W1,W2 and W6 match the first, second and third preconditions, respectively. The bindings resulting from these matches are

```
(PRODUCTION MAKE-TEAM
      (GOAL ^NAME CREATE-TEAM)
      (EMPLOYEE ^NAME <N1> ^PREVIOUS-PROJECT <P> ^EXPERTISE HARDWARE)
      (EMPLOYEE ^NAME <N2> ^PREVIOUS-PROJECT <P> ^EXPERTISE COMPILERS)
⟶
      (MAKE TEAM ^FIRST-MEMBER <N1> ^SECOND-MEMBER <N2>))
```

(a)

```
W1:(GOAL ^TYPE CREATE-TEAM)
W2: (EMPLOYEE ^NAME A ^PREVIOUS-PROJECT WARP ^EXPERTISE HARDWARE)
W3: (EMPLOYEE ^NAME B ^PREVIOUS-PROJECT WARP ^EXPERTISE HARDWARE)
W4: (EMPLOYEE ^NAME C ^PREVIOUS-PROJECT PSM ^EXPERTISE HARDWARE)
W5: (EMPLOYEE ^NAME D ^PREVIOUS-PROJECT PSM ^EXPERTISE HARDWARE)
W6: (EMPLOYEE ^NAME E ^PREVIOUS-PROJECT WARP ^EXPERTISE COMPILERS)
W7: (EMPLOYEE ^NAME F ^PREVIOUS-PROJECT WARP ^EXPERTISE COMPILERS)
W8: (EMPLOYEE ^NAME G ^PREVIOUS-PROJECT PSM ^EXPERTISE COMPILERS)
W9: (EMPLOYEE ^NAME H ^PREVIOUS-PROJECT PSM ^EXPERTISE COMPILERS)
```

(b)

Figure 2.1: A simple production system: (a) a production rule, and (b) working memory.

consistent: for example, both the second and third preconditions contain the variable <P>, and the matches to W2 and W6 each result in binding <P> to WARP. This successful match causes the rule to fire, creating a new tuple using the bindings established by the precondition matches:

```
(TEAM ^FIRST-MEMBER A ^SECOND-MEMBER E)
```

Production systems are relevant to my work for two reasons. First, one can envision representing quantified clauses as productions and implementing quantified subsearch as something like production match. Production match and firing is very like the computation needed to detect a contradiction between a quantified clause and a set of ground assignment literals: each clause literal would have to be matched to a ground literal, and the resulting bindings checked for consistency. For example, let us think of the production in Figure 2.1 as a quantified clause, i.e.

$$c : (\neg\text{GOAL(CREATETEAM)} \vee \neg\text{EMPLOYEE(<N1>,<P>,HARDWARE)} \vee$$
$$\neg\text{EMPLOYEE(<N2>,<P>,COMPILERS)} \vee \text{MAKETEAM(<N1>,<N2>))}$$

and suppose the tuples to be atoms made TRUE or FALSE in some assignment, e.g.

$$\begin{aligned} A = \quad & \{\langle\text{GOAL(CREATE-TEAM)}, \text{TRUE}\rangle, \\ & \langle\text{EMPLOYEE(A,WARP,HARDWARE)}, \text{TRUE}\rangle, \\ & \langle\text{EMPLOYEE(B,WARP,COMPILERS)}, \text{TRUE}\rangle\} \end{aligned}$$

Leaving aside for the moment the question of what clause c actually means, consider the potential role to be played by the production system in this scenario. Viewed in the context of this mini FQSAT problem, the production shown in Figure 2.1 performs a computation similar to unit propagation. In this case, the three atoms match the three preconditions of the original rule, causing the rule to be fired. But *this firing is precisely the kind of inference used in unit propagation*: all but one of the literals in a clause are falsified, so we are required to add the remaining literal to the current assignment. As we shall see, the core matching process behind productions can be used to model other subsearch functions, too.

The second reason Production Systems are relevant to the current work is because one can draw upon considerable past research into efficient incremental algorithms for maintaining production states over time. In both DPLL and production system runs, the current state changes slowly: small numbers of assignments or tuples, respectively, are added at each iteration. Therefore, incremental algorithms that can change the minimum state possible, avoiding wholesale recomputations, are desirable.

2.1.1 The Rete Algorithm

In particular, I adapt two incremental Production System algorithms in this dissertation. The first is the Rete algorithm [22], used in the main JOINSAT algorithm, which incrementally maintains the bindsets that make up the state of a join network. The second is the Collection Match algorithm [2], used in an optimized version of JOINSAT, which also tracks bindsets, but which uses an efficient compressed representation for them. I will present my own version of the Collection Match algorithm in section 3.4.1.

In this section, I examine the Rete algorithm, whose information flow is pictured in Figure 2.2 , reprinted from [2]. The figure shows a Rete network designed to implement the production from Figure 2.1 (a) that I examined above. In the figure, the symbols looking rather like bins full of tuples (the tuples from Figure 2.1 (b)) are called *memory stores*. The top three bins, called *alpha memories*, are where the active tuples enter the system: each bin services a precondition, holding those active tuples that match with it. Typically this matching is performed by a discrimination net for the entire production system, so that multiple productions sharing the same precondition need not do the same matching work twice.

Each tuple is made into a (singleton) set of tuples called a *token*, and the tokens are passed downwards towards *beta nodes* (boxes with x's in them in the figure), which match pairs of tokens from their left and right inputs iff the tokens' variable bindings are consistent, as discussed earlier in this section. Tokens that are matched together are unioned together and stored in *beta memories*, which also appear as bins in the figure. In this production, the beta node connected to the first two alpha memories simply makes matches of all its inputs, because the first two preconditions share no variables. However, the second beta node matches upon the variable <P>, so some tokens cannot be matched together, e.g. {W1,W2} cannot be matched with {W8}, because W2 binds <P> to WARP, while W8 binds <P> to PSM. The tokens created by the second beta node are used to fire the production.

The primary benefit of Rete is its incrementality: the production's memory stores are updated by adding new tuples at the appropriate alpha memories and matching them with those tokens already in the system. For example, if a new tuple (call it W10) were added to the second alpha memory, it would immediately match with W1 and form a new token {W1,W10} in the first beta memory. This token would in turn be passed to

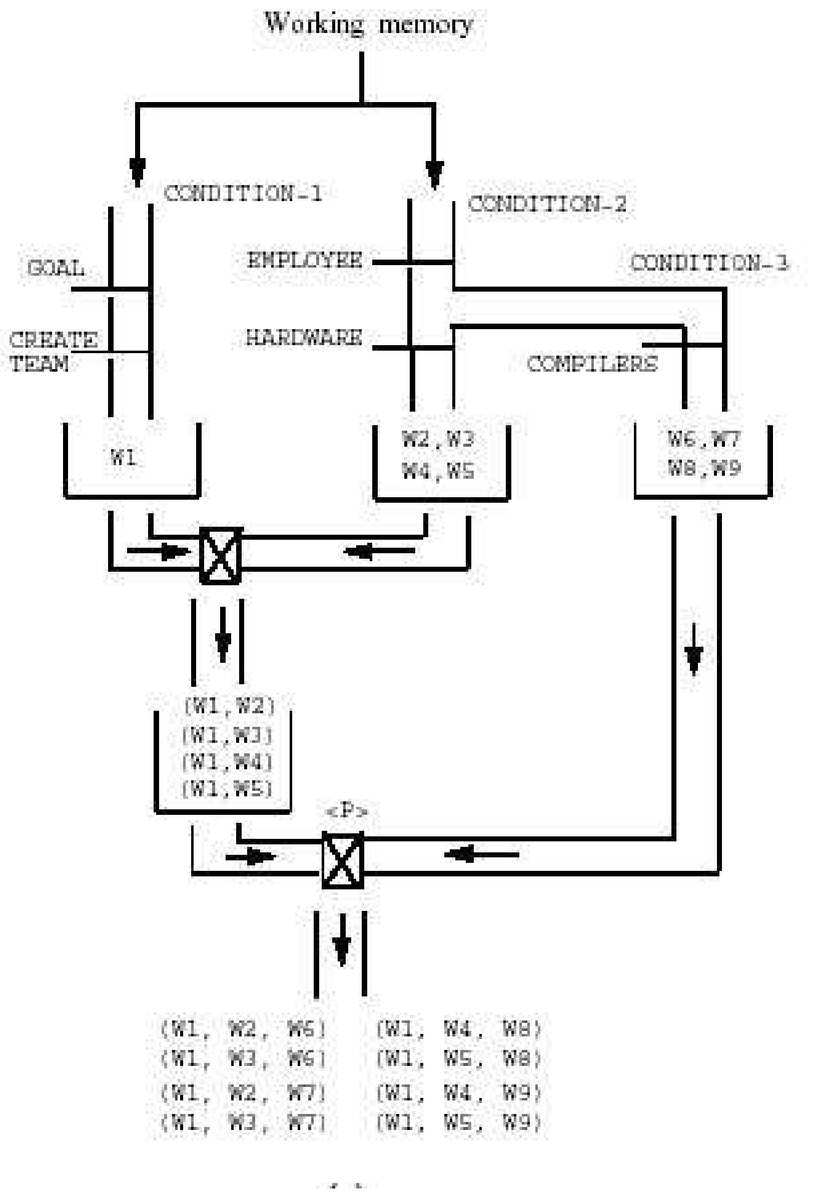


Figure 2.2: Example of the Rete network.

the second beta node and be matched against any compatible tokens in the third alpha memory, and so on. Deletion takes place similarly, a tuple being deleted at the top, prompting a cascade of deletions as all tokens containing the tuple are removed. These updates take place without any recomputation of matches not relevant to the tuple being added or deleted.

Now, Rete is designed to deal with tuples, which are structured entities similar to entries in a database table. This is not an entirely natural representation to use when one really wants to model the matching of ground literals with quantified predicates. In addition, much of the terminology associated with Production Systems reflects the field's origin as an offshoot of relational database research. I will therefore introduce entirely new notation in the next chapter: productions become *join networks*, whose properties are more focused toward logical ends, hopefully making them a more elegant tool for my purposes.

2.2 Join Networks and FQSAT Subsearch - A First Look

In this section, I define join networks and their components, and then show how a join network might handle the example in section 1.5.2 more efficiently than a ground instantiation solver like MACE2 does.

A *join network* has the form of a connected, directed acyclic graph whose nodes are *literal nodes* and *join nodes*. A single join network is used to compute subsearch functions for a given quantified clause c. Figure 2.3 shows a join network for the clause c shown in equation 1.5.

Each network is organized into a matrix; network nodes are referenced by their level and position in it, e.g. J_{23} (sometimes written $J_{2,3}$ if there is any possibility of confusion) is the node in the third position from the left in the second level. Note that the network may not have a node in every position of the matrix. The first network level contains one literal node for each literal $l \in c$; each literal node contains the information defining its counterpart literal, e.g. its predicate and arguments. Typically, literal nodes with negated literals are placed before literal nodes with non-negated literals for reasons explained later. I sometimes speak of literal nodes and their literals interchangeably, as in a “negated literal node”.

Subsequent levels contain only join nodes, sometimes called simply *joins* for short. If nodes m and n are connected by an edge $m \rightarrow n$, I say that m is n's *parent* and n is m's *child*. A node (of either type) may have any number of children , but literal nodes never have parents, while join nodes always have exactly two parents. A node that has no child is called a *terminating node* for the network. The network in Figure 2.3 (a) is an example of a *simple join network*, in which every node has exactly one child except for the terminating join node.

To describe how join networks operate, I first introduce the concept of a bindset, the primary datum used by join networks. A *bindset* is a set of variable bindings: the variables are from the quantified clause c, while the bindings are elements of the domain D. An example of a bindset containing two bindings is $\{x = 1, y = 2\}$. Bindsets are assumed to be *coherent*, i.e. a bindset cannot map a given variable to two different domain values. I say that two bindsets are *consistent* or *compatible* if they do not map a given variable to two different domain values; therefore, two consistent bindsets may be unioned together to form a new coherent bindset.

Network nodes perform computation by creating, storing, and passing *entries*, which are really just bindsets bundled with a data pointer or two to be explained later; I sometimes speak of bindsets and entries interchangeably. Entries are created and stored at literal nodes by the following process. Each ground literal l from the current assignment A is matched against any literal node L_{1i} whose literal lit_i has the same predicate as literal l and the opposite sign (e.g. an assignment literal $\neg p(a)$ may match upon a clause literal $p(x)$, but not clause literals $\neg p(x)$ or $q(x)$. If the arguments for l and lit_i unify, an entry is created in L_{1i} containing the bindings created by the unification process (e.g. literal $\neg p(a)$ unifies with clause literal $p(x)$, producing the bindset $\{x = a\}$, but not with clause literal $p(b)$). For convenience in these contexts, I sometimes refer to literals as (negated or non-negated) atoms.

Entries are constantly added and removed from literal nodes as A changes. If an assignment literal and a literal node's clause literal successfully match/unify, I say that the assignment literal *falsifies* the clause literal (and the node), because at least one ground instance of the clause literal is falsified by the assignment literal. I also say that the assignment literal *defines* the resulting bindset/entry. For example, in Figure 2.3 (a), assignment literal l_2 is shown in the bin above literal node L_{12}, signifying that l_2 falsifies L_{12}'s clause literal; the resulting defined entry is e_2, shown below L_{12} to indicate it was created there.

Literal nodes pass pointers to their entries to each of their child joins. Using the pointers, the join registers the entries in *indexes* that are used for more matching; I refer to an entry passed to a join this way as an *input*. Join nodes attempt to match together each input from their left parent with each input from their right parent; two inputs match if their associated bindsets are compatible. When the inputs match, the join creates and stores a new entry whose bindset is the union of the two input bindsets. The join passes pointers to this new entry to each of its children, so matching a new literal with a literal node may cause a cascading chain of entries to be added throughout the network, ending only at the terminating node.

Figure 2.3 shows a join network that computes subsearch for the quantified clause c from section 1.5.2. In the initial state, shown in Figure 2.3 (a), let assignment A be

$$A : \{\langle a_2, \text{TRUE}\rangle, \langle a_3, \text{FALSE}\rangle\} \tag{2.1}$$

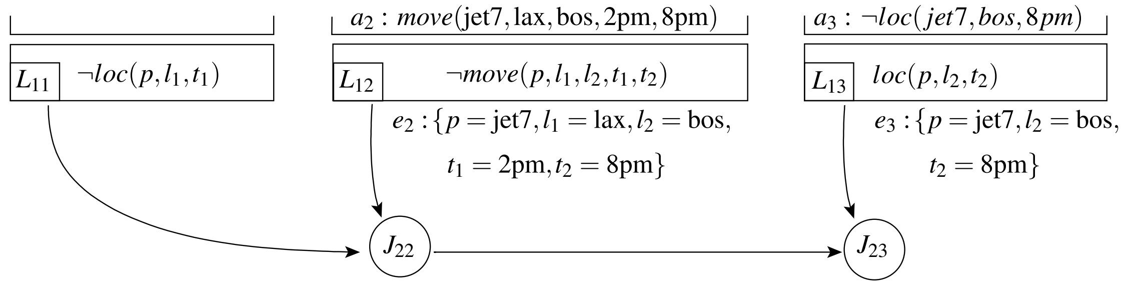
where $a_2 : move(\text{jet7}, \text{lax}, \text{bos}, \text{2pm}, \text{8pm})$ and $a_3 : loc(\text{plane1,bos,8pm})$. Since a_2 falsifies L_{12}'s literal, an entry

$$e_2 : \{p = \text{jet7}, l_1 = \text{lax}, l_2 = \text{bos}, t_1 = \text{2pm}, t_2 = \text{8pm}\} \tag{2.2}$$

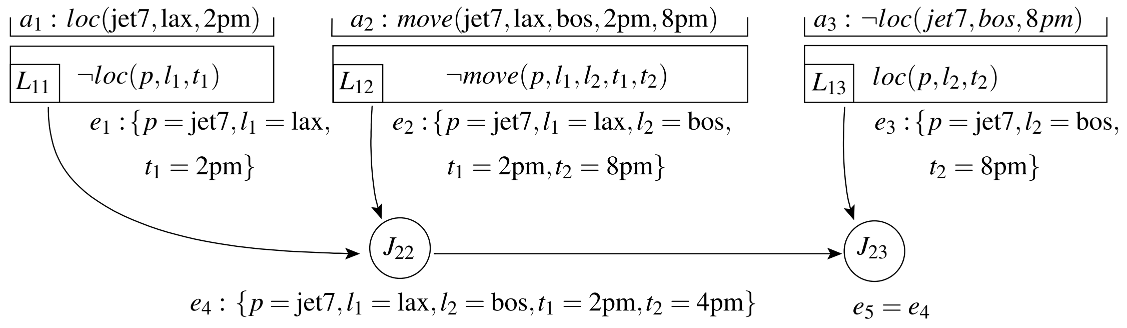
with the resulting bindings is created in L_{12}; $\neg a_3$ similarly defines an entry

$$e_3 : \{p = \text{jet7}, l_2 = \text{bos}, t_2 = \text{8pm}\} \tag{2.3}$$

contained in L_{13}. Initially, clause literal L_{11} is not falsified by any literal. Initially, join J_{22} has (a pointer to) entry e_2 in its index, but the join contains no entries of its own, because its left parent L_{11} has no entries, so no matches can be made. Similarly, because J_{22} is empty, its child J_{23} is empty.


a_2 : $move$(jet7, lax, bos, 2pm, 8pm)
a_3 : $\neg loc(jet7, bos, 8pm)$
L_{11} $\neg loc(p, l_1, t_1)$
L_{12} $\neg move(p, l_1, l_2, t_1, t_2)$
L_{13} $loc(p, l_2, t_2)$
$e_2 : \{p = \text{jet7}, l_1 = \text{lax}, l_2 = \text{bos}, t_1 = \text{2pm}, t_2 = \text{8pm}\}$
$e_3 : \{p = \text{jet7}, l_2 = \text{bos}, t_2 = \text{8pm}\}$
J_{22}
J_{23}

(a)


a_1 : loc(jet7, lax, 2pm)
a_2 : $move$(jet7, lax, bos, 2pm, 8pm)
a_3 : $\neg loc(jet7, bos, 8pm)$
L_{11} $\neg loc(p, l_1, t_1)$
L_{12} $\neg move(p, l_1, l_2, t_1, t_2)$
L_{13} $loc(p, l_2, t_2)$
$e_1 : \{p = \text{jet7}, l_1 = \text{lax}, t_1 = \text{2pm}\}$
$e_2 : \{p = \text{jet7}, l_1 = \text{lax}, l_2 = \text{bos}, t_1 = \text{2pm}, t_2 = \text{8pm}\}$
$e_3 : \{p = \text{jet7}, l_2 = \text{bos}, t_2 = \text{8pm}\}$
J_{22}
J_{23}
$e_4 : \{p = \text{jet7}, l_1 = \text{lax}, l_2 = \text{bos}, t_1 = \text{2pm}, t_2 = \text{4pm}\}$
$e_5 = e_4$

(b)

Figure 2.3: An example of join network operations.

Now suppose in the second state, pictured in Figure 2.3 (b), $\langle a_1, \text{TRUE}\rangle$ is added to A, where

$$a_1 : loc(\text{jet7}, \text{lax}, \text{2pm}) \tag{2.4}$$

Updating the join network when a_1 is assigned TRUE requires the computation of any new entries caused by matching a_1 with the appropriate literal node(s). When atom a_1 is assigned TRUE, it successfully matches $lit_1 : \neg loc(p, l_1, t_1)$, so entry

$$e_1 : \{p = \text{jet7}, l_1 = \text{lax}, t_1 = 2\text{pm}\} \tag{2.5}$$

is added to literal node L_{11}. Tis sets off a chain reaction of matches. Entry e_1 is in turn passed to join node J_{22}, and J_{22} successfully matches entries e_1 and e_2, whose bindsets are consistent. Join J_{22} therefore creates a new entry

$$e_4 = e_1 \cup e_2 = \{p = \text{jet7}, l_1 = \text{lax}, l_2 = \text{bos}, t_1 = 2\text{pm}, t_2 = 8\text{pm}\} \tag{2.6}$$

Join J_{22} passes e_4 to join J_{23}, which matches entries e_4 and e_3, creating entry e_5, which is equivalent to e_4.

2.2.1 Implications of the Example

This short example prompts three observations. First, *match entries represent resolution inferences of (ground) assignment literals against the clause.* Because the literal nodes store bindsets defined by falsifying literals, and because the join nodes match together consistent bindsets, a match entry at a join node J has a special significance. Consider all the literal nodes that are ancestors of J. If we think of these literal nodes as a subclause of the larger clause, then the assignment literals associated with our match entry *falsify* this subclause, in the sense that they falsify some ground instantiation of it. For example, entry e_4 represents a set of assignment literals that collectively falsify $\neg loc(p, l_1, t_1) \vee \neg move(p, l_1, l_2, t_1, t_2)$, a prefix of c. Sets of assignment literals can also be resolved against the original clause c to make inferences. In this case, resolving a_1 and a_2 with c in succession allows us to infer $a_3 : loc(jet7, bos, 8pm)$.

The second observation is that *these inferences can be used to perform subsearch functions.* The above inference can be used for unit propagation, inferring a_3 from the resolution of a_1 and a_2 with c. The same mechanism can be used to identify PFGCs from which to select an literal to branch upon. This process is complex and is explained at length in section 2.4.1, but briefly, if some prefix of the quantifed clause is falsified by some match, then we can apply the bindings contained in the match to the rest of the clause and pick one of the resulting literals to branch upon. A final subsearch function that can be computed using these inferences is testing for inconsistency. Consider the the join J_{23}, which has all clause literals as ancestors. If this join ever makes a successful match, the assignment literals associated with that match collectively falsify clause c, indicating that the current assignment can never be extended to satisfy this clause.

A final observation is that *the work necessary to maintain join networks depends only on those atoms that are actually* TRUE *or* FALSE *in the assignment, not on those atoms that are* UNVALUED. Because so few atoms are actually valued in the current assignment, only two matches are required. In contrast, implementing the same incremental change in section 1.5.2 requires a ground instantiation solver to update all 10,000 clauses containing negated atom $\neg a_1$. The great majority of these clauses may never be satisfied or falsified during the course of problem-solving, so much of this effort on the solver's part may be unnecessary.

Now, if each literal of c was falsified by many assignment literals, one might expect its join network to create a great many matches, possibly making matching more expensive than MACE2-DPLL's ground clause updates. But for this particular domain, a given assignment literal l_1 falsifying literal node L_{11} is unlikely to create more than one match in join J_{22}, because of the underlying semantics of the domain. Of all the assignment literals l_2 that falsify the second literal, $\neg move(p_1, l_1, l_2, t_1, t_2)$, only one should match l_1's bindings. This is because a plane starting at a given place and time will never make two or more simultaneous moves to new places and times; only one move is possible in the real world.

The foregoing is an example of the *negative bias* (mentioned in section 1.7.3) shown by DPLL-style solvers: the expectation that models for problems of interest will have a high ratio of FALSE to TRUE atoms. In general, I claim that join networks benefit from the negative bias in two ways:

1. Negated problem literals will be matched against relatively few positive atoms, because most atoms, if assigned at all, are assigned FALSE by a model (and any assignment explored while finding a model).

2. Even non-negated literals will not encounter many matches, because typically a model is found before most atoms are valued at all by the assignment. This is true for two reasons. The first is that typically a DPLL-style algorithm need only value some subset of the Herbrand base $\mathscr{H}(C,d)$, the set of all ground atoms constructable from the (function-free, in this case) clauses and the domain values in D, in order to find a model. The second is that the negative bias allows a solver to solve clauses without explicitly assigning their negated literals to FALSE (as seen in MACE2-DPLL, figure 1.7, line 4). Another way of putting this is that the negative bias contributes to an overall *unvalued bias*: an expectation that model-finding does not require very many atoms from the Herbrand base to be explicitly valued.

The negative and unvalued biases also drive my organization of join networks so that negated literals come before non-negated literals in the network. To see why, one must reference database query optimization, specifically, the technique of *join ordering* [24]. It is well known that a given database query consisting of joins between several tables can take significantly more or less time depending upon the order that the joins are executed. We should expect to see similar phenomena in a join network, because each join network is essentially a query that selects sets of falsifying literals from the literal nodes they falsify, with joining occuring to ensure that the falsifying bindings are compatible. In this analogy, the network literal node assumes the role of the database table.

If join networks are like queries, then as with query optimization, we generally want to avoid joining a “large table” (a literal node having many entries) early in the sequence of joins. It is often far better to join that literal node last, when presumably the number of entries joined with it will be very small, as these entries are a result of matching multiple successive joins. The join of this small number of matches with our presumptive large literal node should then produce as few matches as can be hoped for. Therefore, it makes sense to join together negated literal nodes first, because according to the negative and unvalued biases, typically they will be falsified by fewer literals than will non-negated literal nodes. Figure 2.4 shows an example of this phenomenon. The figure shows data

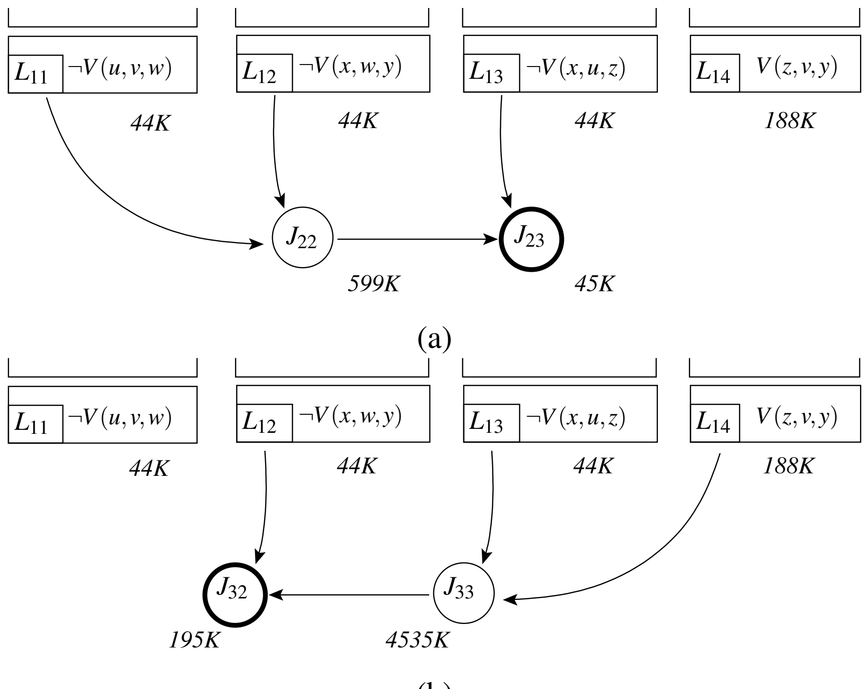


Figure 2.4: Increased matching from early joining of a large table.

from actual runs of the o1e1 problem presented in section 3.1.1. The two join networks in the figure are annotated with the number of entries created at each node. The join network in part (a) flows left to right, joining negated literals first, while the network in part (b) flows right to left, joining a non-negated literal first. As expected, network (b) creates many more entries than network (a) because of the matching occurring at join node J_{33}, where the large table (non-negated literal node L_{14}) is first joined upon.

2.3 A Complexity Comparison of MACE2 and Join Network Subsearch

Integrating a quantified, join-based subsearch system into the larger DPLL algorithm involves a number of technical challenges and resulting elaborations. The complexity comparison between such an elaborated system and MACE2 becomes murky. However, a useful starting point for this comparison is to compare the basic incremental operations of MACE2 with those of the basic join-based system sketched out in section 2.2.

2.3.1 Initial Cost Estimate for MACE2

Let us examine an arbitrary quantified clause c that MACE2 instantiates in the INSTANTIATE procedure, such that $c = (Pr_1(\overrightarrow{x}) \vee Pr_2(\overrightarrow{x}) \vee \cdots \vee Pr_n(\overrightarrow{x}))$, in which each Pr_i is some negated or non-negated predicate instance containing some subset of the variables in c. Assume clause c has m clause variables, i.e.$|\overrightarrow{x}| = m$, and suppose that the current domain size is d. Then the set G_c of distinct ground instantiations of c will have size d^m. Now, during execution of the search algorithm, suppose a literal l is added to the assignment such that l matches $P_1(\overrightarrow{x})$, i.e. $\neg l$ is a grounded version of $P_1(\overrightarrow{x})$. Now, P_1 only contains some subset x' of the set $\overrightarrow{x}$ of all variables in the clause c; suppose that $x' = n$. This means there must be d^{m-n} ground clauses in G_c that match literal l.

Now, recall from section 1.7.3 that MACE2 keeps tallies for each ground clause, and that these tallies must be updated every time a flipped literal matches some clause literal. Therefore, when literal l is flipped, MACE2 must update a number of clauses exponential in $m-n$. Also significant is that the number of clauses updated will be proportional to the domain size d.

2.3.2 Initial Cost Estimate for Join Networks

A join network's complexity is fundamentally different from MACE2's. The work done by MACE2 is primarily a function of the syntactic complexity of the input clauses, that is, how many variables each contains. In contrast, the join networks found in the JOINSAT algorithm operate by matching literals from the current assignment with each other. JOINSAT therefore has complexity dependent on the size of that assignment and the matching factor, or degree to which the literals match together.

I should first note that a typical assignment is usually far smaller than the Herbrand base. This is because of the factors mentioned in section 2.2.1, i.e. that the solver usually finds a solution before all atoms are assigned, and that most atoms given an assignment are assigned FALSE.

Given a quantified clause c, JOINSAT uses a network similar to the simple join network in Figure 2.3 to compute various subsearch functions via matching. There are various dominant costs required to update the join network: creating entries, inserting them into join indices, checking for matches, etc. However, since these costs are all linearly proportional to the total number of successful matches made by the network, we may assess the network's complexity using this latter measure. For simplicity in the following calculation, I drop the level designator for each node, so e.g. join J_{22} becomes J_2, literal node L_{11} becomes L_1, and so on.

At any join J_i, the number of matches can be expressed by

$$M(J_i) = Left\_Inputs(J_i) * Factor(J_i) * Right\_Inputs(J_i) \tag{2.7}$$

where $Left\_Inputs(J_i)$ is the number of inputs from J_i's left parent, $Right\_Inputs(J_i)$ is the number of inputs from the right parent, and $Factor(J_i)$ is the matching factor, or probability that any two arbitrary bindsets match at J_i. Now, consider that J_i's right parent is always a literal node L_i, and that the number of inputs from L_i is exactly the number of literals matching L_i: I denote this number of matches $M(L_i)$. Furthermore, the number of inputs from J_i's left parent is $M(J_{i-1})$. Therefore, we obtain the recurrence:

$$\begin{array}{rcccccc} M(J_i) & = & Left\_Inputs(J_i) & * & Factor(J_i) & * & Right\_Inputs(J_i) \\ & = & M(J_{i-1}) & * & Factor(J_i) & * & M(L_i) \\ M(J_2) & = & M(L_{11}) & * & Factor(J_i) & * & M(L_2) \end{array} \tag{2.8}$$

The above recurrence can be consolidated to:

$$M(J_i) = \left(\prod_{k=2}^{i} (Factor(J_{k-1}) * M(L_k)) \right) * M(L_{11}) \tag{2.9}$$

To simplify further, let us introduce some representative members. Given an array of similar entities (e.g. $M(J_1) \ldots M(J_n)$), I define a representative member as a member of the array that can be substituted into a Big-O equation for any of the other members without falsifying the resulting equation . Obviously, to meet this restriction, the most costly member from the array must be used. First letting the array be the set of $Factor(J_i)$ for all joins, let $Factor$ be the most costly matching factor of these. Then letting the array be the set of $M(L_i)$ for all literal nodes, let M_L be the most costly of these. Using these measures, the above equation can be converted into a simplified Big-O equation:

$$M(J_i) = \mathcal{O}(Factor M_L)^i \tag{2.10}$$

.

Using equation 2.10, an upper bound on the cost of the entire matching process can be set. The overall cost is dominated by the join match cost:

$$Cost(J_n) \quad = \quad \mathcal{O}(\textstyle\sum_2^n M(J_i)) \tag{2.11}$$

Substituting in the equivalence in equation 2.10:

$$\begin{aligned} Cost(J_n) &= \mathscr{O}(\textstyle\sum_2^n M(J_i)) \\ &= \mathscr{O}(n * M(J_i)) \\ &= \mathscr{O}(n * (Factor * M_L)^n) \end{aligned} \tag{2.12}$$

At this stage, I note that M_L, the number of matches of our representative literal L, should be expressable in terms of d. The total number of ground instances of L is d^p, where p is the number of distinct clause variables in L. Therefore M_L is certainly less than d^p. If the number of ground instances of L that *are not* in the current assignment A is expressed as d^l, M_L may be expressed as d^{p-l}:

$$Cost(J_n) = \mathscr{O}(n * (Factor * d^{p-l})^n) \tag{2.13}$$

Now, we should also be able to express the representative matching factor $Factor$ in terms of d. The matching factor $Factor(J_i)$ is determined by the number of joined variables at a given join J_i. If only one variable is joined, and there are d possible bindings, one would expect that a given left input and right input would only match with probability $\frac{1}{d}$. Letting j be the number of variables joined at our representative join, $Factor$ may be expressed as $\frac{1}{d^j}$. Substituting this value into equation 2.13, we obtain:

$$Cost(J_n) = \mathscr{O}(n * (d^{p-l-j})^n) \tag{2.14}$$

which may be finalized as

$$Cost(J_n) = \mathscr{O}(n * d^{n*(p-l-j)}) \tag{2.15}$$

From this equation it is evident that like MACE2, JOINSAT's complexity is in fact exponential, but in the number of clause literals n and the number of variables p in a representative predicate. Like MACE2, JOINSAT's complexity measure is proportional to the domain size d.

At first glance, an analyst might expect JOINSAT's complexity to be greater than MACE2's. Both measures have a base of d, but JOINSAT's exponent is a product of two measures, whereas MACE2's is simply the number of clause variables v. In addition, JOINSAT's exponent contains a multiplier of n. However, there are several reasons to expect that JOINSAT's cost in practice will be less than that of MACE2. First, at worst, JOINSAT's exponent can be no greater than $n * p$, which should not be much larger than v. To see this, consider that $n * p$ can be seen as the total number of variable instances (including duplicate instances) in some representative quantified clause, while v is the total number of variables (this time excluding duplicate instances) in the same quantified clause. Even if each clause variable appears three times in the clause, the resulting advantages for MACE2 should be more than counterbalanced by the effect of the negative exponents l and j.

Let us examine these exponents individually. In most joins, one would expect the number of joined variables j to be almost as large as the number of predicate variables p. This is because the number of join variables is constantly increasing as the join network moves rightward. This is especially true in the later joins, in which the left input bindsets already contain most of the clause variables. Even in a worst-case join matching relatively early, one would still expect at least one variable to be matched upon.

Now let us appraise the influence of l, remembering that d^{p-l} is the number of literals in the current assignment matching our representative clause literal. Now, if l were 0, then d^{p-l} would become d^{p}, meaning that every possible instantiation of our representative clause literal would be present in the current assignment.. But as stated in section 1.5.2, in practice this is unlikely: most problems can be solved by small assignments/models, so one would expect l to be close in value to p. If the above arguments are correct , $p - l - j$ should be very small, perhaps less than 1, and JOINSAT's cost should be far less than MACE2's.

2.4 Design Challenges of Integrating Join Networks into an FQSAT Solver

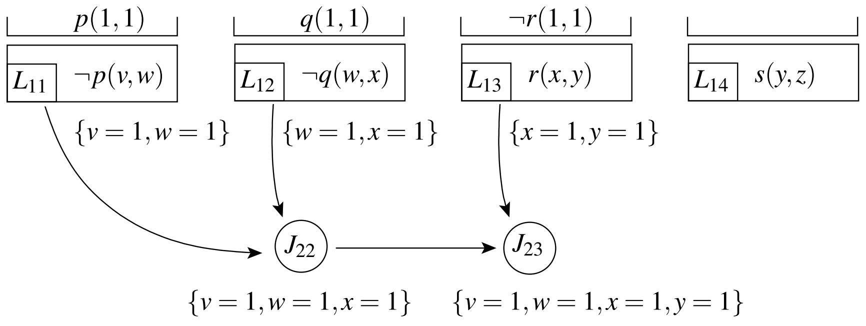
Section 2.2 sketched out how a join system could perform subsearch functions like detecting clause falsification and unit propagation literals. However, the full integration of a join system into a DPLL-style algorithm involves several other technical challenges that must be solved. I examine these solutions in this section before examining the full algorithm in detail in section 2.5.

2.4.1 Ensuring Completeness Via Literal Choice

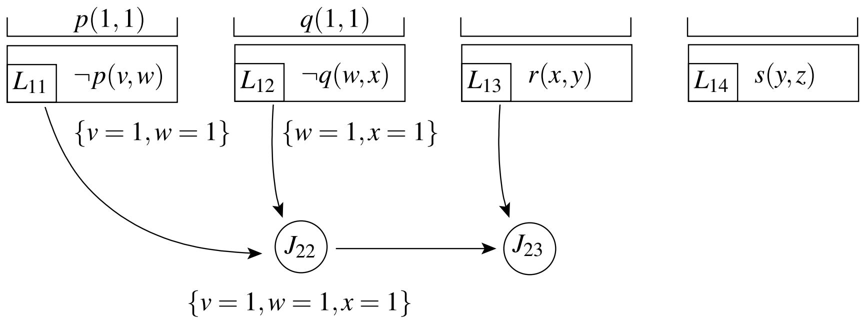
As seen in section 1.7.3, MACE2 chooses which literal to flip next in two ways: unit propagation and selecting positive literals from unvalued clauses. One advantage of MACE2's exponential representation is that finding eligible clauses for these processes is straightforward: each ground clause is examined and various tallies determine if the clause is a candidate for propagation or branching. While the join network in section 2.2 seems to be capable of computing these same functions, in the general case this computation is more complicated, because a simple sequence of joins does not produce all the information contained in MACE2's system of ground clauses and tallies.

Figure 2.5, which reuses the example problem from section 1.7.3, illustrates several situations that are problematic for join networks. Figure 2.5 (a) portrays a join network for the clause $\neg p(v,w) \vee \neg q(w,x) \vee r(x,y) \vee s(y,z)$ that has been updated to represent search space node ⑥, in which $r(1,1)$ is flipped to FALSE. In this case, a match is created at join J_{23}, indicating that the first three literals of the clause are falsified by the literals associated with the match. This sets up an opportunity to perform unit propagation by resolving the literals against the clause. The problem in this case is that the output of resolution is a literal that is not ground, but quantified: $s(1,z)$. This case seems relatively easy to solve: we just instantiate $s(1,z)$ into all possible ground literals, e.g. $s(1,1)$and $s(1,2)$, and immediately flip both of these to TRUE.

However, literal selection can produce less straightforward cases. Consider Figure



(a)



(b)

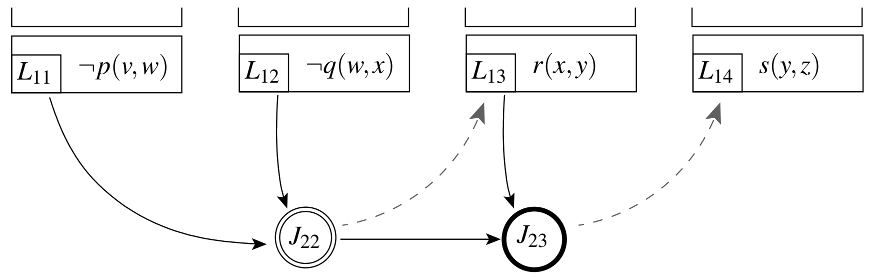
Figure 2.5: JOINSAT design challenges.

2.5 (b), which shows search space node ②, in which the first two literals of clause c are falsified. The set of partially falsified ground clauses defined by these bindings is represented by

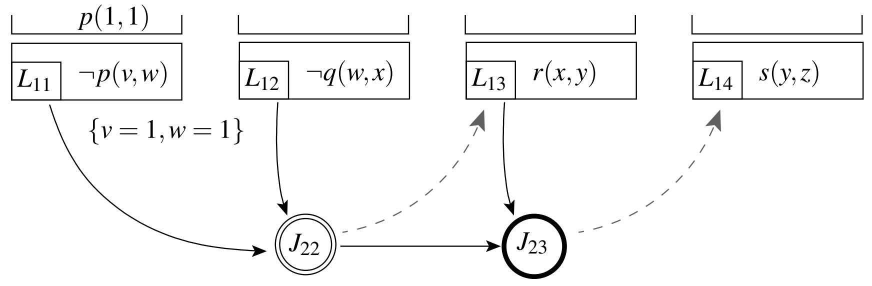
$$\neg p(1,1) \vee \neg q(1,1) \vee r(1,y) \vee s(y,z) \tag{2.16}$$

To ensure completeness in this case, the algorithm needs to implement some form of DPLL's line 6 (from Figure 1.1), selecting ground literals that satisfy these ground clauses and branching upon them. One problem is that there are multiple literals that could be chosen: any instantiation of $r(1,y)$ or $s(y,z)$ flipped to TRUE would satisfy some ground clause subsumed by the formula in equation 2.16. Another problem is that unlike in the unit propagation case, the process of selecting and flipping all the ground literals necessary to satisfy these PFGCs takes place over many iterations of the DPLL procedure. So, some systematic mechanism is needed to track the literals we have already flipped and to determine which literals remain to be flipped. For example, having chosen and flipped literal $r(1,1)$, some mechanism is needed to remind us *not* to choose $r(1,1)$ again, but instead to choose a new literal, say $r(1,2)$, that satisfies other PFGCs implied by equation 2.16. Finally, suppose the search path proceeding from flipping $r(1,1)$ to TRUE ends in a backtrack. It is clear that (1) $r(1,1)$ must next be flipped to FALSE, and (2) doing so prompts the propagation of the ground literals subsumed by $s(1,z)$, just as occurred in Figure 2.5 (a). However, the question remains of how to coordinate all these actions in a fashion that preserves the systematicity (and therefore completeness) of the overall search.

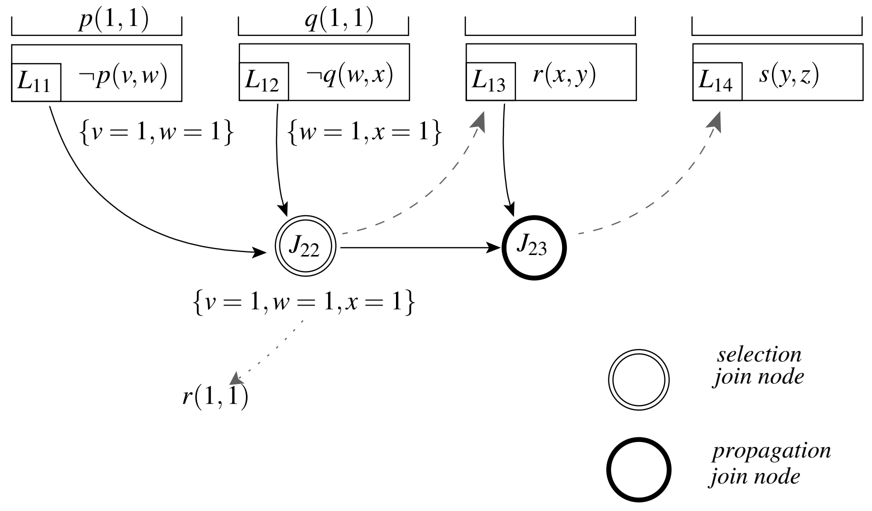
I use Figure 2.6 to illustrate my solution to these problems, as well as illustrate all of JOINSAT-DPLL's operations, in the next section. The solution resembles a waterfall, an often-used idiom in computer science. I first define two special classes of join nodes: *selection join nodes* and *propagation join nodes*; these are shown in the figures as a double circle and a bolded circle, respectively. In a simple join network having m negated literals and n non-negated literals, joins $J_{2,m} \dots J_{2,m+n-2}$ will be selection join nodes, while join $J_{2,m+n-1}$ will be a propagation join node. Therefore, the first selection node joins the last negated literal node with the rest of the network, while the single propa-



(a)



(b)



(c)

Figure 2.6: Operations of the JOINSAT network (part 1).

gation node is a termination node joining the penultimate literal node with the network. For example, in Figure 2.6 (a), join node J_{22} is a selection node joining the last negated literal node, L_{12}, while join node J_{23} is a propagation node joining the penultimate literal node, L_{13}.

Entries in selection and propagation nodes have the special function of storing pointers to literals that are candidates for flipping. These literals are computed by referencing the *literal node target* of the selection/propagation node. In Figure 2.6, literal node targets are shown using dashed arcs: for example, the selection node J_{22}'s literal node target is L_{13}, and the propagation node J_{23}'s target is L_{14}. When an entry is added to the selection/propagation node, that entry's bindset is applied to the literal of the node's literal target node. The resulting (possibly non-ground) literal represents the set of ground literals that will be flipped to TRUE or FALSE in order to satisfy the clause.

For example, in Figure 2.6 (c), the flipping of $q(1,1)$ causes an entry $e_1 : \{v=1, w=1, x=1\}$ to be added to selection node J_{22}. Node J_{22}'s literal node target is L_{13}, so e_1's bindset is applied to $r(x,y)$, producing the literal $r(1,y)$. This literal defines a vector of ground atoms that are subsumed by it: in this case, with domain size $d=2$, this vector is $\langle r(1,1), r(1,2) \rangle$. I label this vector the *target vector* of entry e_1 and order the vector using each literal's (ground) arguments, i.e. $r(1,1)$ precedes $r(1,2)$ because $11 < 12$. While e remains active, i.e. until we backtrack and remove $q(1,1)$, e_1 maintains a pointer to the least atom in the target vector that is unvalued by the current assignment. I call this atom the *least unvalued target* of the entry. In Figure 2.6 (c), atom $r(1,1)$ is unvalued, so initially it is the least unvalued target of entry e. The atom itself keeps a list of *update entries*, entries for which it is a least unvalued target.

During the unit propagation and literal selection phases, propagation nodes and selection nodes, respectively, are scanned for candidate atoms. Each entry in these join nodes contains a least unvalued target; the set of all such targets system-wide is the set of candidates for flipping. When a target is flipped, it no longer is unvalued, so it updates all the entries on its update entries list. Continuing the example, in Figure 2.6 (d), the selection phase chooses entry e's target for flipping, and $r(1,1)$ is assigned TRUE.

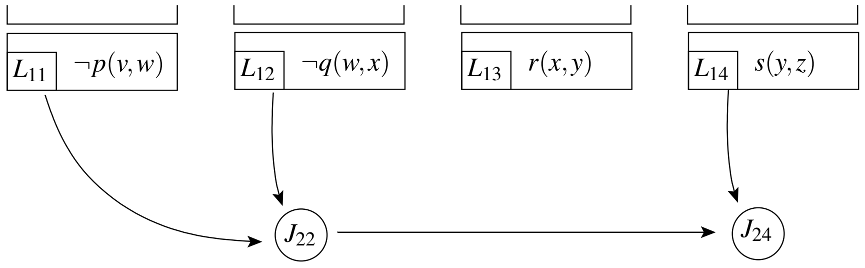


Figure 2.7: A second simple join network for propagation.

The atom's update entries of course include entry e. Entry e's least unvalued target is therefore updated to the next lowest atom in its target vector, $r(1,2)$.

Over time, this system prompts JOINSAT to give all target atoms of an entry e the truth value required to satisfy all the PFGCs defined by entry e. However, in the nature of things, the search process may be forced to assign some of these targets the opposite truth value instead. For example, consider Figure 2.6 (e), in which backtracking forces JOINSAT to revise $r(1,1)$'s assignment to FALSE. In this case, those PFGCs that were satisfied by $r(1,1)$ now must be satisfied by flipping some instantiation of $s(y,z)$ to TRUE. This is automatically accomplished by the interaction of the join network with the target system described above. In this case, the negated atom $\neg r(1,1)$ falsifies clause literal $r(x,y)$, so the negated atom is added at literal node L_{13}. This in turn creates a match entry $e_2 : \{v=1, w=1, x=1, y=1\}$ at propagation node J_{23}. Since J_{23} is a propagation node, entry e_2 is immediately given a least unvalued target of $s(1,1)$, and this target is flipped to TRUE during unit propagation (in search space node 7, shown in Figure 2.6 (f)). Flipping $s(1,1)$ to TRUE updates e_2's least unvalued target to $s(1,2)$, and this atom is correspondingly made TRUE via propagation. At this point, $s(1,2)$ is the last atom in e_2's target vector, so e_2's least unvalued target is now set to ϕ.

This sequence illustrates the waterfall-like behavior of the atom selection system. In this analogy, the PFGCs, represented by entry bindings, are the water. Some of the PFGCs are satisfied at a particular join by flipping entry targets, but those that are not create new matches, thus flowing on to a join further on in the network, where they are addressed anew.

2.4.2 Unit propagation

Section 2.4.1 showed the mechanism by which JOINSAT added literals to satisfy PFGCs, ensuring completeness. However, even using the waterfall system, the simple join network illustrated in Figure 2.6 is inadequate for computing all possible opportunities for unit propagation. This network can indeed compute unit propagation for the last clause literal, because any bindset output from join J_{23} represents a ground instantiation of literals $L_{11} \ldots L_{13}$ that is falsified by the current assignment. However, unit propagation cannot be similarly calculated for, say, literal L_{13}. The bindsets coming into join J_{23} only represent ground literals that falsify the first three clause literals, as opposed to clause literals 1, 2, and 4, which is what would be needed to compute a propagated unit for literal node L_{13}. Of course, this could be achieved by using a second simple join network in which the order of the joins is rearranged, as in Figure 2.7. However, this approach would require n networks for an n-literal clause, which seems extravagant. It would be desirable to formulate one join network of smaller size that computes unit propagation for all clause literals.

Let us reflect upon the requirements of a join network that computes unit propagation for a given literal lit whose literal node is L_{1i}. This join network must terminate in a join node J that will output the bindings allowing us to construct the propagated unit. Therefore, to perform valid unit propagation inference, J's ancestors must include every literal node *except* L_{1i}. Indeed, the key difficulty is the requirement that J must not have L_{1i} as an ancestor, because this makes it difficult for us to use the same join structure to compute unit propagation for multiple clause literals.

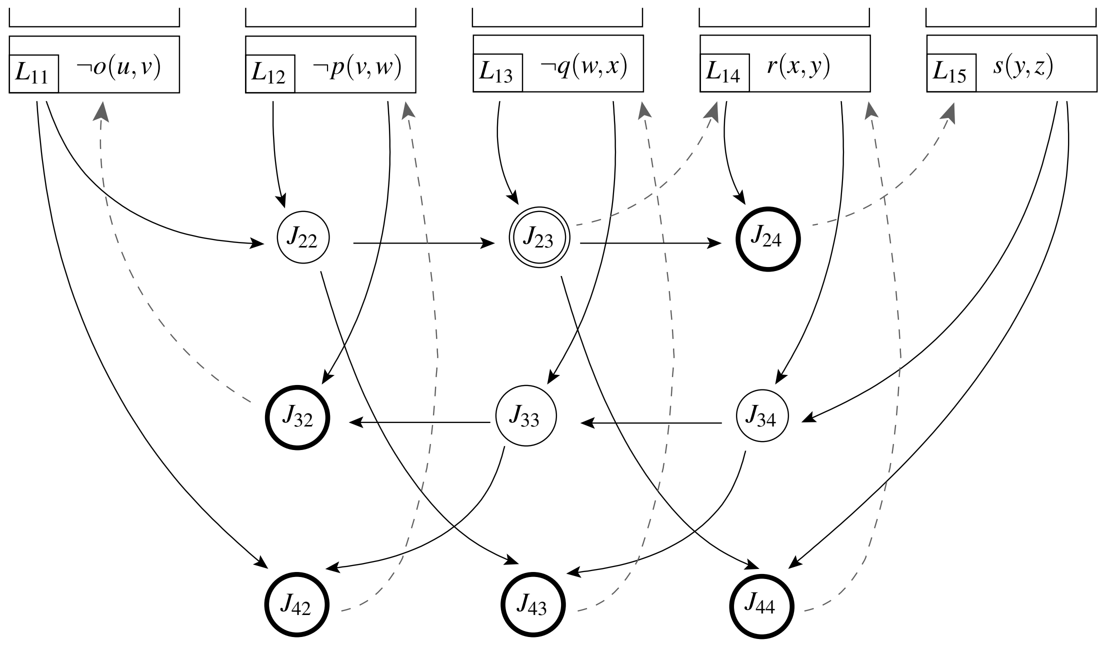


Figure 2.8: JOINSAT unit propagation network

2.4.2.1 The unit propagation network

Figure 2.8 depicts my solution, the join network actually used by JOINSAT to compute unit propagation and also literal selection. The network consists of the literal node layer plus three layers of join nodes. The first join layer forms a simple join network flowing left to right. The last of these join nodes, node J_{2n-1}, is a (terminating) propagation node whose target is literal node L_{1n}. The second layer proceeds right to left; join node J_{32} is here a (terminating) propagation node having literal node target L_{11}. The final layer joins together inputs from the first two layers and contains the bulk of the propagation nodes for the system. These nodes have as targets literal nodes $L_{12} \ldots L_{n-1}$.

It is easy to see that nodes J_{2n-1} and J_{32} perform valid unit propagation inference for L_{11} and L_{1n}, respectively. Each of these joins is the terminating join of a simple network that includes each literal node except for the join's literal node target. Verifying valid inference for the joins in the third join level is only slightly more involved. Consider a given third-level join J_{3i}, which computes unit propagation for literal node L_{1i}. Now, J_{3i}'s left and right parents are J_{1i-1} and J_{2i+1}, respectively. We know that J_{1i-1} has as ancestors all node literals to its left, i.e. $L_{11} \ldots L_{i-1}$, while J_{2i+1} has as ancestors all node literals to its right, i.e. $L_{i+1} \ldots L_{1n}$. Therefore, join J_{3i} has as ancestors all literal nodes except for L_i.

Therefore, this network computes unit propagation for each clause literal, yet contains less than $3n$ joins, where n is the number of clause literals. This is a great improvement on the n^2 joins that would be required by using n different simple join networks to compute propagation. The other important feature of this network is that it incorporates the left-to-right simple join network (including selection join nodes) used in the prior section for the selection phase; this simple join network forms the first join level (joins $J_{21} \ldots J_{2n}$) of the larger network.

2.4.2.2 Adding derived clauses

In an obscure but significant case, the above network fails to compute available propagation opportunities. This case is sometimes found in quantified clauses having two separate instances of the same predicate, and is best illustrated with an example. Consider the clause

$$c : \neg p(u,v) \vee \neg q(v,w) \vee \neg q(w,x) \vee r(x,y) \vee s(y,z). \tag{2.17}$$

Suppose that the current assignment A contains the following literals: $p(1,2)$, $\neg r(2,3)$ and $\neg s(3,4)$. Now, one of the ground instantiations of c is

$$g : \neg p(1,2) \vee \neg q(2,2) \vee \neg q(2,2) \vee r(2,3) \vee s(3,4) \tag{2.18}$$

and if the assignment literals are resolved against g we obtain

$$\neg q(2,2) \vee \neg q(2,2) \tag{2.19}$$

Therefore, this is a unit propagation opportunity, because $\neg q(2,2)$ is implied by the current assignment. MACE2 will detect this opportunity, because it keeps an explicit representation of clause g and its various tallies, but JOINSAT will not, even using the full network described in the previous section. This is because that network detects propagation by joining together all literal nodes except one. But in this case, we would need to join together all literal nodes except *two*: literal nodes L_{12} and L_{13}. So, this case will not be detected.

There are several potentially good solutions to this problem. Ours is to create new clauses, called *derived clauses*, in which pairs of problematic clause literals like those found in c are unified together. These derived clauses do not replace the originals, but rather are added to the original clause set. In the foregoing example, the procedure DERIVED-CLAUSES unifies the second and third literals of c, producing the new literal $\neg q(w,w)$ and associated replacement bindings $y = z$ and $z = w$. A new clause c' is then created from c, with the new literal replacing the second and third literals and the replacement bindings applied to the entire original clause:

$$c' : \neg p(u,v) \vee \neg q(w,w) \vee r(x,y) \vee s(y,z). \tag{2.20}$$

The new clause c' will detect the unit propagation opportunity shown earlier.

Adding derivative clauses can be a costly solution, as it creates wholly new clauses which require new join networks to compute subsearch for them. Furthermore, if the original clause contains n identical literals, as many as 2^n derived clauses might need to be created. In practice, derived clauses add less cost than might be expected because the new clauses often have literals that are rarely matched, such as $q(w,w)$ in the example. However, since I find empirically that derived clauses are often not worth their cost, I do not construct them for clauses in which a great number must be created.

2.4.3 Finding clauses with k unvalued literals

There remains the more general problem of detecting partially falsified ground clauses having k unvalued literals mentioned in section 1.5. This function is used by MACE2 to implement an optional heuristic literal selection function (see the code for MACE2-SELECT-LITERAL in Figure 1.10). For a join-based subsearch system like JOINSAT, solving this problem is probably so difficult as to be infeasible. To see this, suppose we have a quantified clause c of size n and we choose a set K of size k of its literals. Suppose further that we want to detect the partially falsified ground instantiations of c having k unvalued literals, each of which is an instantiation of some quantified literal in K. Now, any clause in this set has the following two properties. First, all its literals *are not* instantiated from K must be negated by the current assignment. Second, all its literals that *are* instantiated from K must be unvalued. Computing the set of clauses having the first property seems feasible: a join network having as ancestors all the literals not in K would be required. Detecting clauses having the second property is much harder, to the point that I only sketch how it might be achieved.

First, for each literal in K, we would need to compute the set of unvalued atoms with the same predicate. For example, for literal $p(x,y)$ and domain size $d=3$, if the current assignment A contains two assignments, e.g. $A : \{\langle p(1,1), \text{TRUE}\rangle, \langle p(1,2\rangle, \text{FALSE}\}$, then the set of all ground instantiations of $p(x,y)$ that are not in A is

$$\{p(1,3), p(2,1), p(2,2), p(2,3), p(3,1), p(3,2), p(3,3)\} \tag{2.21}$$

As stated before, it is expected that the number of unvalued literals will be very large compared with the number of literals assigned a value, so one would expect this computation to be very expensive. If these matches were computed for each unvalued literal, they would still need to be joined together to ensure that the bindings for each literal were consistent. Then this network would need to be connected to the network computing the first property .

This is already quite complicated, but the real reason this computation is infeasible is the great number of possible sets K of unvalued literals. If a clause c has n literals, there are $\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n!}{k!(n-k)!}$ different subsets of c with k literals, and presumably each of these would require the kind of complicated network sketched above. So, it appears that the general problem of finding PFGCs with k unvalued literals is beyond JOINSAT's powers.

Happily, the efficiency gain from this heuristic is comparatively small. Most modern solvers implement literal choice heuristics associated with clause learning, and these typically tally the number of literal occurrences in conflict clauses, as opposed to finding clauses with k unvalued literals.

2.4.4 Clause subsumption

I noted in section 1.7.2.6 that MACE2 eliminates subsumed ground clauses from its clause set in preprocessing. It is unclear whether this task is feasible for JOINSAT, precisely because it does not instantiate all the ground clauses. For example, in MACE2, a ground unit clause $c_1 : \neg p(1)$ could resolve with $c_2 : p(1) \vee \neg q(1,1) \vee r(2)$, producing the clause $c_3 : \neg q(1,1) \vee r(2)$, which then subsumed $c_4 : \neg q(1,1) \vee r(2) \vee s(3)$. Suppose c_2 and c_4 have quantified counterparts used by JOINSAT : $c'_2 : p(w) \vee \neg q(w,x) \vee r(y)$ and $c'_4 \neg q(w,x) \vee r(y) \vee s(z)$. This ground reasoning cannot be duplicated using joining alone: we could insert $\neg p(1)$ into the join network for c'_2, but this would just produce an entry, not a ground clause.

Of course, we could use the resolution of c_1 and c'_2 to create a new clause $c''_2 : \neg q(1,y) \vee r(z)$, and then use subsumption reasoning in reference to c'_4 to show that a specialization $c''_4 : \neg q(1,x) \vee r(y) \vee s(z)$ is subsumed by c''_2. But this does not eliminate c'_4 itself, so it achieves little. Because I cannot see a way to make clause subsumption profitable, JOINSAT does not implement it.

2.4.5 State of the art SAT optimizations

As noted in section 1.3.2, recently a number of important optimizations have improved the performance of SAT solvers by several orders of magnitude. For the join network method to be of real research interest, it must be able to support the implementation of these optimizations in a quantifed setting. Happily, and in contrast to some of the above sections, join networks appear to compute enough information to make these optimizations feasible. For example, conflict learning typically analyzes information from a backtracking point to make a new ground rule. This information consists of a list of assignments that together directly take part in the conflict. Join networks can compute these assignments by looking at the structure of entries that caused the propagation of two conflicting atoms. For example, if an atom $p(1)$ is propagated, a tree of entries can be traced back, one per node, from the entry that had $p(1)$ as a least unvalued target. This tree of entries ultimately leads to entries at literal nodes that can be traced to assigned atoms. Non-chronological backtracking requires the same information.

One potential problem is that adding many ground conflict clauses will make the overall problem being solved propositional rather than quantified. Although join networks can solve pure SAT problems (in this case the literals in the literal nodes simply have no variables), there is reason to believe they will be less efficient than ground solvers at doing so. Therefore, the degree to which adding ground conflict clauses might bog down a join network is a an open question. However, one could always create a hybrid system that used ground methods for the conflict clauses and join networks for the original, quantified clauses.

State of the art heuristics for atom choice revolve around how recently a particular conflict-created clause was used to prune search; this information is no more difficult to track using networks than using a ground solver.

```
procedure JOINSAT($\mathscr{T}, D$)
1: $\mathscr{T}_1$ := EXISTENTIALSTOFUNCTIONS($\mathscr{T}$)
2: $\mathscr{T}_2$ := FLATTEN($\mathscr{T}_1$)
3: $C_1$ := FUNCTIONCONSTRAINTS($\mathscr{T}_1, D$)
4: $C_2$ := SYMMETRYCONSTRAINTS($\mathscr{T}_1, D$)
5: $QC_1$ := QUANTIFIEDCNF($\mathscr{T}_2$)
6: $QC_2$ := DERIVED-CLAUSES($QC_1$)
7: $S$ := JOINSAT-INITIALIZE($QC_1 \cup QC_2, C_1 \cup C_2$ )
8: JOINSAT-FLIP($S, dummy(), \text{TRUE}$)
9: return JOINSAT-DPLL($S$)
```

Figure 2.9: JOINSAT. Given an FQSAT problem $(\mathscr{T}, D)$, to compute JOINSAT($\mathscr{T}, D$):

2.5 Presentation of the Full Algorithm

In the prior section I explored the primary design challenges to creating a DPLL-style algorithm that uses join networks to compute subsearch. Having described the solutions to these challenges in detail, I can now begin a straightforward presentation of the full algorithm, whose top-level procedure shown is in Figure 2.9. For this purpose, some material introduced earlier is reused, including the example problem used to detail the MACE2-INITIALIZE and MACE2-DPLL algorithms in section 1.7.3; this problem is displayed in Figure 1.5. Figure 2.6 from section 2.4 is also reused by extending the figure to parts d,e,f and g below . Note that in this figure only levels one and two of the larger JOINSAT network presented in Figure 2.8 are displayed; I have constructed the example so that the relevant activity occurs only in these levels.

2.5.1 Preprocessing

Like MACE2, JOINSAT works with logical clauses, but unlike MACE2, these clauses are quantified. Therefore, JOINSAT uses only some of the preprocessing procedures used in MACE2 and detailed in section 1.7.2. In particular, EXISTENTIALSTOFUNCTIONS, FLATTEN, FUNCTIONCONSTRAINTS, SYMMETRYCONSTRAINTS and QUANTIFEDCNF are used without alteration (lines 1-5 in Figure 2.9).

Originally, JOINSAT utilized versions of FUNCTIONCONSTRAINTS (line 3) and SYMMETRYCONSTRAINTS (line 4) that produced quantified clause versions of these constraints. However, it was found that join networks offered no efficiency advantage for such constraints over MACE2, and indeed that the additional overhead of join networks in this case caused a slowdown. Therefore, JOINSAT runs a process of MACE2-DPLL in tandem with its own JOINSAT-DPLL solver; this process performs subsearch solely for the functional and symmetry constraints. Accordingly, JOINSAT uses the MACE2 versions of these functions to produce propositional clauses and also calls MACE2-INITIALIZE to create data structures for them. However, for simplicity these operations are not discussed further in this chapter; instead, the fiction is maintained that the propositional constraints are initialized with the quantified clauses (line 7).

Because JOINSAT uses quantified clauses, MACE2's INSTANTIATE and MACE2-INITIALIZE processes are unneeded, with the exception noted above. The same is true of the CLAUSESPLIT process: generally speaking, JOINSAT's memory usage is not exponential in relation to problem size, so using CLAUSESPLIT offers no benefit. Indeed, by declining clause splitting, JOINSAT's search space is substantially reduced in some cases, offering an advantage over MACE2.

DERIVED-CLAUSES (line 6) was described in section 2.4.2.2. These derived quantified clauses are added to the original quantified clause set. To avoid generation of huge derived sets, JOINSAT generates derived clauses only for those original problem clauses containing 12 clause variables or less.

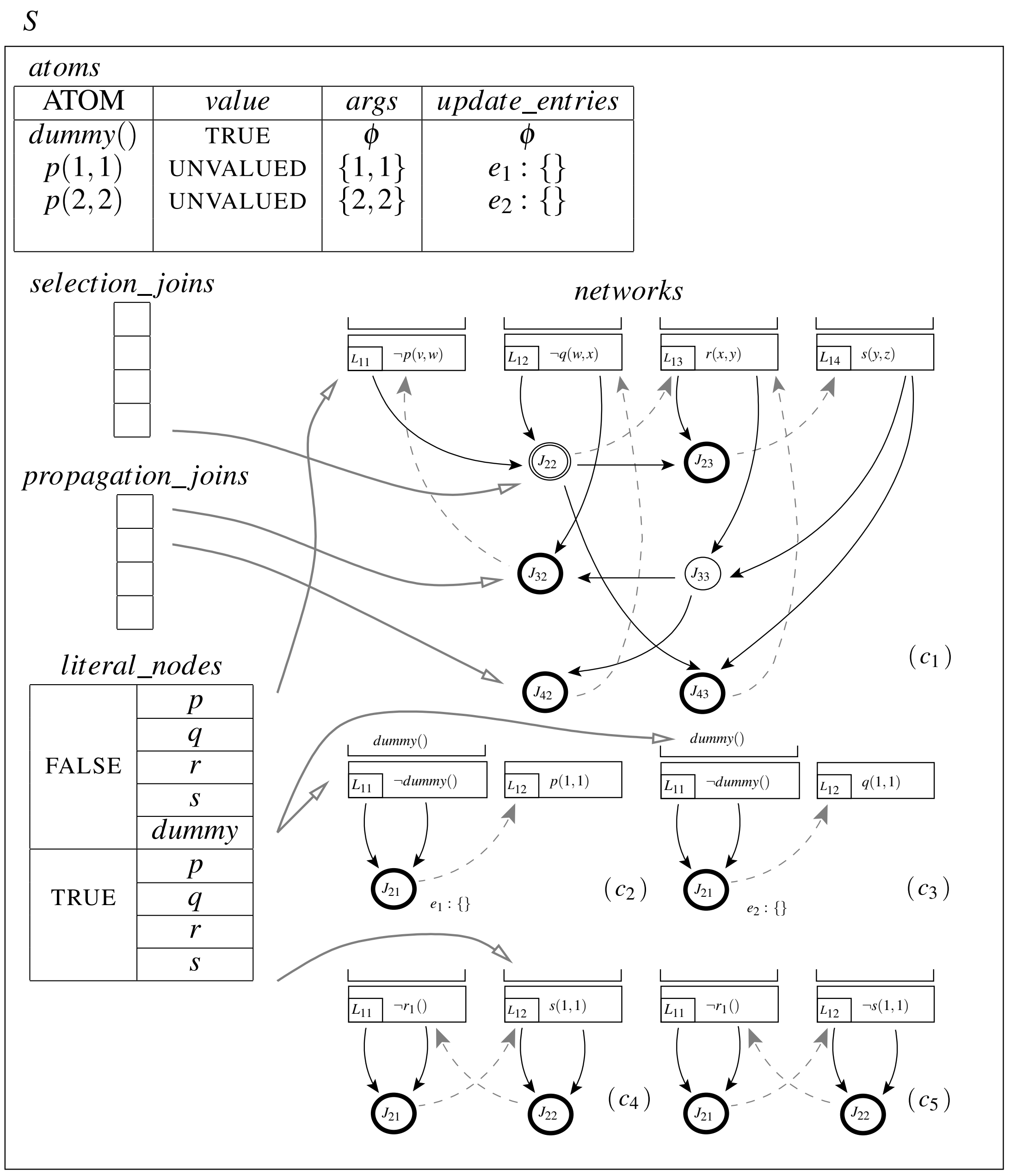
2.5.2 The JOINSAT-INITIALIZE Procedure

In line 7, JOINSAT calls JOINSAT-INITIALIZE, which creates the initial state structure *S*, pictured in Figure 2.10. The structure *S* contains two main elements: *atoms* and *networks*. The element *atoms* is a table, initially empty, containing entries representing ground atoms, similar to the table used in MACE2-DPLL. As with that table, an atom is represented in *S.atoms* if the atom is valued in the current assignment, and in this case *atom.value* contains one or more truth values. However, an atom can also be present in the table if the atom is the least unvalued target of some join entry, in the sense introduced in section 2.4.1; such atoms are obviously not valued by the current assignment. Indeed, atoms typically are created and inserted into *S.atoms* in such a capacity, and only later are assigned some truth value. Each *atom* represents its corresponding ground atom using the fields *predicate*, which contains the atom's predicate (e.g. p for atom $p(1,1)$), and *args*, a vector of the domain element arguments in the atom (e.g. $\langle 1,1\rangle$ for atom $p(1,1)$).

The second element in S, *S.networks*, contains a join network for each quantified clause in QC'; each network computes unit propagation and literal choice for that clause. Except in the case of very short clauses (≤ 2 literals), each network has the form of the network introduced in Figure 2.8. However, in the example problem, clauses 2-5 are relatively short, so their networks are simpler: I do not detail their operations in any detail in the example, instead focusing on clause c_1.

In the case of clauses having only one literal, we want that literal to immediately be unit propagated. However, the target method described in section 2.4.1 implementing propagation and selection does not work for clauses of length one: the method requires some join to point to the desired target and catalyze propagation, but no obvious join exists here. I therefore use a special-purpose network for such clauses[1], for example, for clause 2 of the example problem, which happens to be ground: $p(1,1)$. First, a dummy literal $\neg dummy()$ is added at the front of the clause, and then an extremely simple network having only two joins ($J_{11}$and J_{12}) is made. Each join has the literal node on its "side" as its left and right parent, and each join is a propagation join having the literal node on the other "side" as its *target_literal_node*. The effect of this network is to make adding either one of the literals immediately cause unit propagation of the other literal. In this case, the $dummy()$ literal has a special status: in line 8 of JOINSAT, a special initial flip of the atom $dummy()$ is made, to TRUE. This flip will cause $p(1,1)$

[1]This rather inefficienct network could certainly be improved upon



atoms

ATOM	*value*	*args*	*update_entries*
$dummy()$	TRUE	ϕ	ϕ
$p(1,1)$	UNVALUED	$\{1,1\}$	$e_1 : \{\}$
$p(2,2)$	UNVALUED	$\{2,2\}$	$e_2 : \{\}$

Figure 2.10: JOINSAT initial state structure S.

to be unit propagated at the start of the JOINSAT-DPLL algorithm. The network for the third clause, $q(1,1)$, has an identical network to the second. I defer discussion of JOINSAT-FLIP until the next section.

Clauses having only two literals also use a simplified network, being too short to require complex selection and propagation networks. The simplified network is the same as that used for one-literal clauses, but the clause is not supplemented with the *dummy*() literal. The networks for the fourth and fifth clauses, which each have two literals, are of this kind.

Literal nodes have fields (similar to those found in an *atom*) that represent their corresponding literal. The *predicate* field represents the literal's predicate (p in the case of $p(1,x)$) and *args* is a vector containing the domain elements and clause variables found in the literal ($\langle 1,x \rangle$ in the case of $p(1,x)$). Each join node contains an *index*, an indexed table containing data structures called *entries*. As explained in 2.2, entries are passed to the join by its left and right parents and matched together using *index*. The table is indexed upon two values: first, LEFT or RIGHT, denoting whether the entry was passed by the left or right parent, and second, by a subset of the bindings found in the entry's *bindset*. In the actual implementation of JOINSAT, *index* is implemented as a dynamic hash table that converts these bindings into a hash key.

The remaining data in S are lists or arrays of pointers to various classes of nodes in *S.networks*. *S.literal_nodes* is the most complex, a two-dimensional array of pointers to all literal nodes in *S.networks*. This array is indexed by two quantities: the *predicate* of the literal node and the literal node's *value*, expressed as TRUE or FALSE. For example, literal node L_{11} from the fourth network has a *predicate* of r and a value of FALSE (because its literal, $\neg r(1,1)$, is negated). As we shall see, these dimensions allow JOINSAT-DPLL to select precisely the set of literals that must be accessed when a given atom is flipped. *S.selection_joins* and *S.propagation_joins* are lists of pointers to all selection and propagation joins, respectively, found in *S.networks*; these lists are used to find candidates for selection and unit propagation. Selection and propagation joins contain fields *target_literal_node*, a pointer to the target node discussed in section

2.4.1. Some other data fields found in the various structures introduced in this section are best explained below.

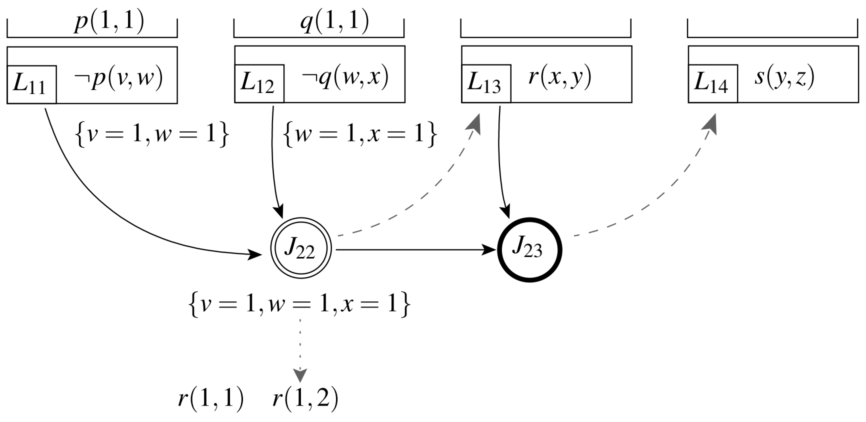
2.5.3 JOINSAT-DPLL

Now that the data structures used to represent the search state have been set out, the JOINSAT-DPLL algorithm (shown in Figure 2.11) and its handing of the example problem can now be presented in detail.. The top-level structure of JOINSAT-DPLL is quite similar to DPLL and MACE-DPLL; most differences concern how tests are conducted and literals selected. As with MACE2-DPLL, each recursive call to JOINSAT-DPLL has its own top-level state variable argument; this variable (i.e., the entire state) is copied for each recursive branch. This is a presentational simplification, as every operation performed by the actual algorithm, including backtracking, is made by incrementally changing the state. In the example, JOINSAT-DPLL begins at search node ⓪.

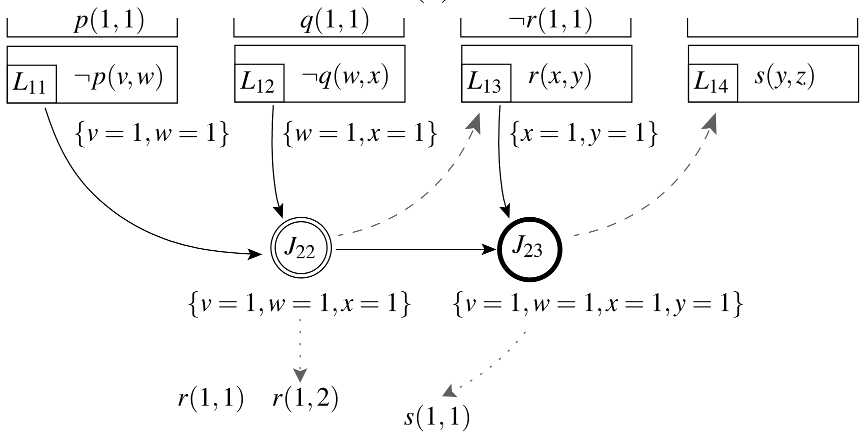
Line 1 of JOINSAT-DPLL performs unit propagation. In the topmost invocation of JOINSAT-DPLL, propagation will include any clause containing the $dummy()$ literal which was flipped in JOINSAT-INITIALIZE. In the example, this includes clauses c_2 : $\neg dummy() \vee p(1,1)$ and c_3 : $\neg dummy() \vee q(1,1)$.

JOINSAT-UNIT-PROPAGATE , shown in Figure 2.12, loops until either no more literals are found to propagate (tracked by the local variable *continue*) or the current assignment has become inconsistent; these checks are performed in line 2. As with MACE2-Unit-Propagate, the inconsistency check is made by scanning all entries in *S.atoms* to verify that no entry has a *value* of both TRUE and FALSE. As explained in section 2.4.1, the unit propagation system tracks propagation opportunities using the entries in propagation joins. So, literals to propagate are found by scanning the *entries* of each propagation join in *S.networks*; each entry has a pointer, *least_unvalued_target*, to an unvalued atom in *S.atoms* (lines 5-7). The truth value to assign is obtained from the propagation join itself (*target_literal_node_value*, in line 8) , and JOINSAT-FLIP the atom in line 9. The loop variable *continue* is set to true if any atoms were flipped.

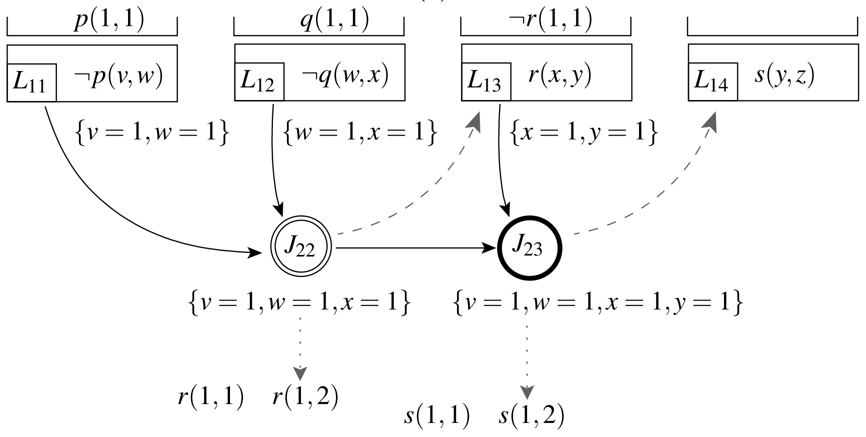
In the example, propagation joins in the networks for clauses c_2 and c_3 each contain



(d)

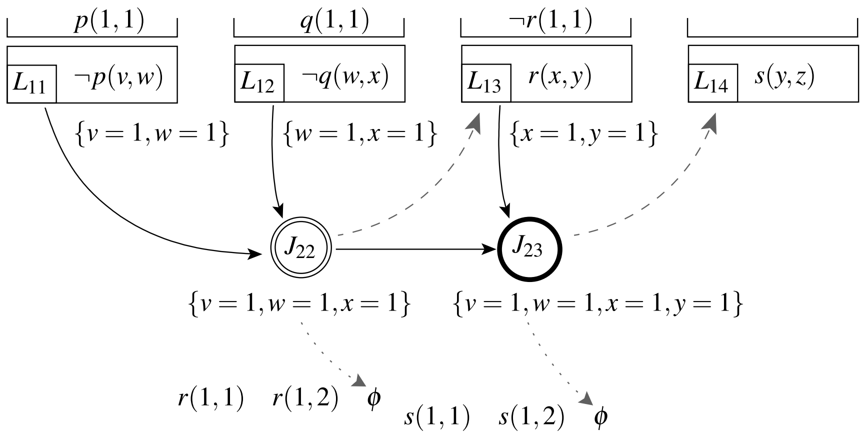


(e)



(f)

Figure 2.6: Operations of the JOINSAT network (part 2).



(g)

Figure 2.6: Operations of the JOINSAT network (part 3).

one entry; these entries' *least_unvalued_target*'s are $p(1,1)$ and $q(1,1)$, respectively. Each join's *target_literal_node_value* is TRUE, so these two atoms are each flipped to TRUE.

At this stage, I examine JOINSAT-FLIP (shown in Figure 2.13) in more detail. Like MACE2-Flip, JOINSAT-Flip begins by updating the current assignment (line 1), pushing the passed truth *val* onto the *value* of the flipped *atom*. However, in JOINSAT-Flip the passed *atom* argument is not an index to *S.atoms*, but instead a pointer pointing directly to the atom record in *S.atoms*. This method is used because unlike MACE2-DPLL, I do not explicitly represent (or index) every possible ground atom in *S.atoms*, but rather, only those that have been assigned a truth value or that are the *least_unvalued_target* of some join entry.

JOINSAT-FLIP's next task, in lines 2-3, is to update the *least_unvalued_target* of any entries that currently have *atom* as their *least_unvalued_target*. Now that the atom has been assigned a truth value, it is no longer unvalued, so the entry is updated. Each *atom* stores a list *update_entries* of pointers to these entries, so for each one INCREMENT-LEAST-UNVALUED-TARGET is

```
procedure JOINSAT-DPLL($S$)
1: JOINSAT-UNIT-PROPAGATE($S$)
2: if for some $atom \in S.atoms$,$atom.value$ contains two inconsistent bindings then
3:     return FAILURE
4: if forall $join \in S.selection\_joins$, forall $e \in join.entries$, $e.least\_unvalued\_target = \phi$ then
5:     return SUCCESS
6: $atom$ := JOINSAT-SELECT-LITERAL($S.select\_joins$)
7: for $val$ in {TRUE,FALSE} do
8:     $S$':= copy $S$
9:     JOINSAT-FLIP($S'$,$atom$,$val$)
10:    if DPLL($S'$) == SUCCESS then
11:        return SUCCESS
12: return FAILURE
```

Figure 2.11: JOINSAT-DPLL.

called. After all entries are updated, *update_entries* is reset (line 4). In the example, when $p(1,1)$ and $q(1,1)$ are flipped, entries having them as *least_unvalued_target*'s are updated, but I will wait for a more interesting instance to examine this process in depth.

In lines 5-21, JOINSAT-FLIP updates the join network by matching literals with literal nodes (lines 5-8) and computing the resulting match entries in joins (lines 9-21). In line 5, the procedure attempts to match *atom* against literal nodes having the same *predicate* and the opposite sign. The *atom* matches an *lnode* if their respective args (vectors of variables and domain elements) unify together. In the example, the only literal nodes having the same predicate as and opposite sign from $p(1,1)$ is node L_{11} in clause c_1. In this case, $L_{11}.args = \langle v,w\rangle$ and $p(1,1)$'s args are $\langle 1,1\rangle$, so unification succeeds, creating a *bindset* of $\langle v=1, w=1\rangle$. In lines 7-8, a new literal entry e_1 is made for this bindset and inserted into a local variable stack called *entries*. The *entries* stack

```
procedure JOINSAT-UNIT-PROPAGATE($S$)
 1: $continue$:= TRUE
 2: while $continue$ = true and for each $atom \in S.atoms$, $atom.value$ does not contain
    two inconsistent bindings do
 3:   $continue$:= FALSE
 4:   for all $prop\_join \in S.propagation\_joins$ do
 5:     for all $entry \in prop\_join.entries$ do
 6:       if $(entry.least\_unvalued\_target \neq \phi)$ then
 7:         $atom := entry.least\_unvalued\_target$
 8:         $val := entry.node.target\_literal\_node.value$
 9:         JOINSAT-FLIP($S, atom, val$)
10:         $continue$:= TRUE
```

Figure 2.12: JOINSAT-UNIT-PROPAGATE(S).

is used to direct the joining process (lines 9-21). Elements are popped off the stack and added to join indices; any resulting matches cause new entries to be created and pushed onto the stack.

In line 10-11, the topmost entry on the stack is popped into a local variable *entry* and a pointer *node* is set to the node that created the entry (node L_{11} in the case of entry e_1). In lines 12-14, *entry* is inserted into the *index* of every join that is a child of *node*; in the case of e_1 there are two such joins, J_{22} and J_{32}. In line 13, the bindings necessary to join the entry with entries from the other parent of the join are computed. The process filter_bindset() copies those bindings from *entry.bindset* that contain join variables into a new *bindset*. These bindings of course contain exactly those variables found in both left and right inputs, and are stored in the *joinvars* field of the *join*. In the case of join J_{22}, the set of joinvars is only $\{w\}$, and e_1's bindset is $\{v = 1, w = 1\}$, so the resulting new bindset is $\langle w = 1\rangle$. In line 14, the entry is inserted into *join.index*, which has two keys: first, the join bindset (really just its domain values), and second, which join parent

```
procedure JOINSAT-FLIP($S, atom, val$)
1:  push($val, atom.value$)
2:  for all $entry \in atom.update\_entries$ do
3:      INCREMENT-LEAST-UNVALUED-TARGET($entry$)
4:  $atom.update\_entries := \phi$
5:  for all $lnode \in S.literal\_nodes$[negate($val$)][$atom.predicate$] do
6:      if $bindset$ := unify($atom.args, lnode.args$) then
7:          $entry$ := make_entry($bindset, lnode$)
8:          push($entry, entries$)
9:      while $entries$ is not empty do
10:         $entry$ := pop($entries$)
11:         $node := entry.node$
12:         push($entry, node.entries$)
13:         for all $join \in node.join\_nodes$ do
14:             $bindset2$ := filter_bindset($entry.bindset, join.joinvars$)
15:             insert($entry, join.index[bindset2][node]$)
16:             for all $entry2 \in join.index[bindset2][join.otherparent[node]]$ do
17:                 $bindset2 := entry.bindset \cup entry2.bindset$
18:                 $entry3$ := make_entry($bindset2, join$)
19:                 push($entry3, entries$)
20:         if $node \in S.propagation\_joins$ or $node \in S.selection\_joins$ then
21:             INCREMENT-LEAST-UNVALUED-TARGET($entry$)
```

Figure 2.13: JOINSAT-FLIP($S, atom, val$).

the *entry* is coming from, the left parent or the right parent; entry e_1 of course comes from the left parent.

Lines 15-18 create new matches by iterating though those entries in *join.index* that have the same bindings for join variables as does *entry* but that come from the opposite parent. In the case of entry e_1, there are not yet any entries from the opposite parent, node L_{12}, so nothing happens; the state of the network after this flip, at search state ①, is shown in Figure 2.6 (b). However, when the next flip occurs, of $q(1,1)$, its entry e_2 has the same join variable bindings as e_1, so this loop is activated with $join = J_{22}$, $entry = e_2$, and $entry2 = e_1$. Line 16 creates a new bindset, the union of the bindsets of the two matched entries. A new entry is made for the bindset and pushed onto the *entries* stack. In the case of the $q(1,1)$ flip, the new bindset computed for new entry e_3 is

$$\begin{aligned} e_1.bindset \cup e2.bindset &= \{v=1, w=1\} \cup \{w=1, x=1\} \\ &= \{v=1, w=1, x=1\} \end{aligned}$$

When e_3 is itself popped off the *entries* stack, it is inserted into (the index of) join J_{23}, among others, but since no literals falsify $r(x,y)$ there are no entries to match e_3 with, and the matching process comes to an end.

Lines 19-21 calls INCREMENT-LEAST-UNVALUED-TARGET for those entries produced by propagation or selection joins, and also stores pointers to the entries at these joins. These pointers are used throughout the JOINSAT-DPLL algorithm for both literal choice and success tests, e.g. JOINSAT-DPLL line 4. Since join J_{22} is a selection join, these lines are performed for entry e_3.

The procedure INCREMENT-LEAST-UNVALUED-TARGET has the function of actually has two functions: to create an initial *least_unvalued_target* for a new entry (lines 2-14), and when the target is no longer unvalued, to update *least_unvalued_target* (lines 15-24) by setting it to the next unvalued atom in the entry's target vector (introduced in section 2.4.1). The algorithnm avoids the explicit storage of target vectors; instead, the next atom in the target vector is computed upon demand.

After fetching the target literal node of the entry's parent node in line 1, INCREMENT-LEAST-UNVALUED-TARGET checks in line 2 whether the *entry* is new and has an unset *least_unvalued_target*. In this case, it is necessary to construct the first atom in the entry's target vector. First this atom's *args* are constructed, in lines 3-10. For each *arg* in *lnode.args*, the atom's *args*are populated with a value. If *arg* is a domain value (line 5), it is copied into *args*. If *arg* is a variable bound by *entry*'s bindset (line 7), the binding (domain value) *val* is copied into *args*. If *arg* is some variable not yet bound, a 1 (the least domain value) is inserted into *args*. In line 11, the procedure checks if the atom has already been created in *S.atoms* using its predicate and *args*. If it has not, it is created and inserted (lines 12-13). Finally, *entry.least_unvalued_target* is set to this atom (line 14). Note that the procedure has not yet checked if *atom* is unvalued: this task is performed next, whether the *entry* is new or just being updated.

In the example, because entry e_3 is new, the first atom in its target vector is constructed. In this case, *lnode* is L_{13}, whose *predicate* is r and whose *args* are $\langle x, y \rangle$. Since variable x is present in e_3's bindset, the binding for x in that bindset, 1, is pushed onto the new *args* vector. Since variable y is unbound, another 1 is pushed onto *args*. Therefore, the atom $r(1,1)$ is created, inserted into *S.atoms*, and set as e_3's *least_unvalued_target*.

In lines 15-24, the procedure loops until some unvalued atom in *entry*'s target vector is found or the vector is exhausted. In this loop, the *args* of the existing *least_unvalued_target* are first copied (line 16). Then these *args* are incremented so that they become those of the next atom in the target vector (line 17). This is done by incrementing only those elements in *args* whose counterpart arguments in the target *lnode* are variables not bound by *entry.bindset*; these are the same elements for which line 10 was reached when that part of INCREMENT-LEAST-UNVALUED-TARGET was operative. The positions of these elements are computed at runtime and stored for each join in *join.target_literal_node_unbound_args*. When all such args have the value d_{MAX}, the increment_args() procedure fails, indicating that *entry*'s target vector is exhausted (line 18). Otherwise, a new atom is either found or constructed and *least_unvalued_target* is

```
procedure INCREMENT-LEAST-UNVALUED-TARGET($entry$)
  $lnode := entry.node.target\_literal\_node$
  if $entry.least\_unvalued\_target = \phi$ then
      for $i := 1$ to $lnode.args.size$ do
          $arg := lnode.args[i]$
          if $arg \in D$ then
              push($arg, args$)
          else if $\langle arg, val \rangle \in entry.bindset$ then
              push($val, args$)
          else
              push($1, args$)
      if $!atom := S.atoms[lnode.predicate][args]$ then
          $atom :=$ make-atom($lnode.predicate, args$)
          $S.atoms[predicate][args] := atom$
      $entry.least\_unvalued\_target := atom$
  while $entry.least\_unvalued\_target \neq \phi$ and $entry.least\_unvalued\_target.value = \phi$ do
      $args :=$ copy($entry.least\_unvalued\_target.args$)
      if !increment_args($args, entry.node.target\_literal\_node\_unbound\_args$) then
          $entry.least\_unvalued\_target := \phi$
      else
          if $!atom := S.atoms[predicate][args]$ then
              $atom :=$ make-atom($lnode.pred, args$)
          $S.atoms[predicate][args] := atom$
          $entry.least\_unvalued\_target := atom$
  if $entry.least\_unvalued\_target \neq \phi$ then
      push($entry, entry.least\_unvalued\_target.update\_entries$)
```

Figure 2.14: INCREMENT-LEAST-UNVALUED-TARGET($entry, S$).

updated (lines 21-23, in a repeat of lines 12-14).

The procedure concludes with *entry* either having an empty *least_unvalued_target*, or with that target atom placing *entry* on its *update_entries* list to support JOINSAT-FLIP lines 2-3. In the example, the new atom $r(1,1)$ is unvalued, so lines 15-24 are short-circuited; entry e_3 is added to the $r(1,1)$'s *update_entries*. The join network state at the conclusion of this process, in search state node ②, is shown in Figure 2.6 (c).

We next return to JOINSAT-DPLL, albeit only at line 2, which is somewhat surprising given all the algorithmic ground already covered. Line 2 tests for assignment inconsistency; this test is identical to the one in JOINSAT-UNIT-PROPAGATE, but inconsistency here prompts failure and backtracking. At this point in the example, the assignment is still consistent.

Line 3 tests whether a model has been found. This success test reflects the alternative representation used by JOINSAT. Unlike MACE2-DPLL, which scans all clauses to find those that are unvalued, JOINSAT-DPLL scans all selection joins in *S.networks*. As explained in section 2.4.1, each entry represents one more more PFGCs, requiring us to assign literals to TRUE to satisfy them. So, if no such entries exist, or every entry has an empty *least_unvalued_target*, indicating that all atoms that could satisfy its PFGCs have already been flipped, then the current assignment is a model and SUCCESS is returned. Note that for the current assignment to be a model, these same properties must also be true for all entries of propagation joins. However, JOINSAT-UNIT-PROPAGATE only concludes under these conditions, so they must still hold at this stage and need not be retested. In the example, at this point the entry e_3 has a non-empty *least_unvalued_target*, so the assignment is not a model.

It is worth mentioning that this model test incorporates the same negative bias mentioned in the discussion of MACE2-DPLL in section 1.7.3, although it is implemented differently. Recall that the point of this biased test is to find ground clauses with all negative literals falsified by assignment A and no positive literals yet satisfied by A; if no such clauses exist, then A can be extended to a model. The leftmost selection node

```
procedure JOINSAT-SELECT-NONNEGATED-LITERAL($Selection\_joins$)
1: for all $sel\_join \in Selection\_joins$ do
2:     for all $e \in sel\_join.entries$ do
3:         if $(e.least\_unvalued\_target \neq \phi)$ then
4:             $atom := e.least\_unvalued\_target$
5:             return $atom$
```

Figure 2.15: JOINSAT-SELECT-NONNEGATED-LITERAL(C,P).

in network level 2 computes the first condition because of its role as a terminating node for the subnetwork joining together all of the quantified clauses's negative literals. If no entries for this node exist, then there are no ground clauses that have all their negative literals falsified by the model. The rest of the join network handles the second condition by targeting atoms that will satisfy ground clauses until all entries are exhausted.

In line 6, an atom to branch upon is selected by calling JOINSAT-SELECT-NONNEGATED-LITERAL, pictured in Figure 2.15. This procedure is the converse of the success test described in the prior paragraph, in that an entry from some selection join whose is non-empty is found. This target is of course an unvalued atom, and is returned as the branch atom. In the example, e_3 is the only qualifying entry, so the atom $r(1,1)$ is returned.

Again returning to JOINSAT-DPLL, lines 7-11 implement the branching recursion upon the selected *atom*, first assigning it true, then false. In line 8, the current state is copied, in line 9, the atom is flipped, and in line 10 the recursive call is made. In line 11, SUCCESS is returned if the recursive call was itself successful. Each invocation reaching line 12 returns FAILURE, indicating that exploration of its portion of the search space failed to find a model.

In the example, the current state S is copied to S', and then $r(1,1)$ is flipped to TRUE, bringing us to search state node ③. During the flip, INCREMENT-LEAST-UNVALUED-TARGET is called for entry e_3, and its *least_unvalued_target* is incremented to atom

$r(1,2)$, as shown in Figure 2.6 (d). However, also during this flip, $r(1,1)$ is matched with the literal node L_{11} of both clauses c_4 and c_5, causing a new entry to be made in the propagation join J_{22} in each clause (not shown).

After the flip, a recursive call of JOINSAT-DPLL is made using S'. In line 1 of this invocation, the new entries in clauses c_4 and c_5 cause $s(1,1)$ and then $\neg s(1,1)$ to be unit propagated, bringing us to search space node ⑤. This in turn causes the inconsistency test of line 2 to FAIL (because the atom $s(1,1)$ has two inconsistent *value*'s) and prompt backtracking. The for loop (line 7) in the first invocation of JOINSAT-DPLL (detailed earlier) is then resumed: S is again copied to S', and this time $r(1,1)$ is assigned FALSE, bringing us to search space node ⑥, pictured in Figure 2.6 (e). Two significant results are shown in the figure. First, entry e_3's *least_unvalued_target* (which was incremented earlier only in S', not S) is again incremented to $r(1,2)$. Second,$\neg r(1,1)$ matches with literal node L_{13} of clause c_1, which in turn leads to a new entry e_4 in propagation node J_{23}. Since join J_{23} has literal L_4 as its *target_literal_node*, the *least_unvalued_target* for entry e_4 is set to $s(1,1)$.

Two more invocations of JOINSAT-DPLL are required to solve the problem. The first invocation performs unit propagation on $s(1,1)$, bringing us to search state node ⑦, pictured in Figure 2.6 (f).. The propagation causes the *least_unvalued_target* of entry e_4 to be updated to atom $s(1,2)$. This atom is also propagated, causing e_4's target vector to be exhausted and moving the search state to node ⑧. When the invocation performs atom choice in line 6, entry e_3's *least_unvalued_target*, the atom $R(1,2)$, is chosen, and when this atom is flipped in line 8, e_3's target vector is exhausted (Figure 2.6 (g)). At this point, at search space node ⑨, all join entries have exhausted their target vectors, so when a final invocation of JOINSAT-DPLL is called, its model test succeeds.

2.6 Discussion: Advantages and Disadvantages

In the following sections, I review the competing advantages of JOINSAT and MACE2 that affect their performance.

2.6.0.1 JOINSAT's Matching Algorithm

JOINSAT's matching algorithm, whose complexity is determined in practice by the number of assigned atoms is more efficient than MACE2's ground approach in cases where the number of assigned atoms is small.

2.6.0.2 MACE2's Slim Representation

The bright side for MACE2 of having clearly exponential complexity is that the constant factors in the complexity figure should be very low indeed. Most of MACE2's work consists of the following actions:

1. For each flip, accessing a record for that ground atom indicating which ground clauses are affected by the flip. Records for all all atoms are created at runtime, so these are simple array accesses and list traversals.

2. For each ground clause, updating its tally. There may be many ground clause tallies (also created at runtime) to update, but each requires only an integer increment or decrement.

This simplicity is in contrast to JOINSAT, which requires various expensive dynamic memory operations, including the creation/deletion of atoms, entries and bindsets, as well as the the insertion of atoms into $S.atoms$ and entries into join indices, which grow in size over time.

2.6.0.3 The Unit Propagation Network and Large Tables

JOINSAT's particular join network, while achieving the crucial goal of implementing unit propagation, is also a significant burden. A cursory inspection predicts the network should increase run times by a factor of three, since the network is roughly three times the size of a simple join network connecting all the clause literals together. However, the network's actual impact is more severe. I explain this by referring to the ideas about query optimization and large tables laid out in section 2.2.1.

With these ideas in mind, it turns out that there is a particular class of network literal nodes that may be thought of as particularly "large tables": this is the class of nodes having non-negated functional literals. Recall that functional literals are those literals created when instances of functions are replaced with functional predicates as explained in 1.7.2.2. In that section, I also showed how the functional properties of such predicates are enforced by adding new ground clauses to the theory. To see why this can lead to a "large table", consider a functional literal $l : v(v_5, v_1, v_4)$. Because predicate v is functional, by the one to one and onto properties of functions we know that for a given value of v_5 and v_1 there is exactly one value of v_4 such that $v(v_5, v_1, v_4)$ is true. But given a domain D of size d, that means a total of d^2 instances of literal l are true, while $d^3 - d^2$ instances are false. Just as importantly, functional literals are a definite counterexample to the *implicit bias* of section 2.2.1, in the sense that any assignment satisfying the functional constraints must explicitly flip all instances of the functional predicate to either TRUE or FALSE, instead of leaving them unvalued. The upshot of all this is that any positive functional literal will have $d^3 - d^2$ ground literals that falsify it as the current assignment gets closer to a solution. This is close to as "large" as a literal can get.

Given that positive functional literals are "large tables" and that I want to avoid joining them early in join sequences, it is discouraging to discover that JOINSAT's network is prone to doing exactly that. As explained in section 2.4.3, positive literals are ordered to be at the end of the clause. But as shown in Figure 2.8, level 3 of my join network is a join sequence beginning with the last clause literal. So, if the positive literal is functional, this network does indeed join a large table early in the sequence, and potential inefficiencies of an order of magnitude must be expected . As we shall see, this hypothetical problem becomes a reality for numerous FQSAT problems used by the community.

2.6.0.4 The Unit Propagation Network: Derived Clauses

In addition to using a large network that matches large tables, in order to compute propagation JOINSAT must also construct supplementary derived clauses, as discussed in section 2.4.2.2. Maintaining networks for these clauses can be a significant burden for some problems.

2.6.0.5 Avoiding Clause Splitting

As seen in section 1.7.2.7, MACE2 relies upon clause splitting to reduce the degree of blowup caused by its exponential representation. While JOINSAT's representation is similarly exponential in the number of clause variables, it is more accurate to say that JOINSAT's complexity is proportional to the size of the current assignment. In practice, the current assignment tends to be small, so JOINSAT can run using the unsplit clauses. This makes JOINSAT's search tree smaller, probably by some constant factor.

2.6.0.6 Clause subsumption

As noted in section 2.4.4, JOINSAT cannot implement clause subsumption, unlike MACE2. It probably does some extraneous matching as a result.

Chapter 3

Optimizations

This chapter presents a series of optimizations made to the base JOINSAT algorithm in an effort to optimize its performance; the chapter also reports experimental comparisons of these optimizations with the base algorithm and with MACE2. I defer a more comprehensive empirical comparison between MACE2 and the fastest JOINSAT version (the static analysis version) until the next chapter.

While the base JOINSAT algorithm offers tantalizing theoretical gains, performing fewer matches/inserts than MACE2 performs clause updates to solve some problems, overall speed is poor. There are multiple causes for this lag, but the most significant are the dynamic memory operations discussed in section 2.6.0.2 and the large table problem discussed in section 2.6.0.3. Therefore, my optimization efforts focus on these issues.

Figure 3.1 shows a map of six optimizations made upon the base JOINSAT system. Although many of these optimizations could potentially be combined, in fact each child optimization was built upon its parent, so only six systems are actually tested. Ideally, one would explore a matrix of systems with various combinations of optimizations, but insufficient development time was available. However, one can use these existing results to make an educated prediction of the performance of these optimizations in other configurations.

I begin the chapter by presenting the problems and methodology used in these experiments. Then experimental results for MACE2 and for the base JOINSAT algorithm are presented and discussed. In each of the remaining chapter sections, I explain one of the six optimizations and discuss experimental results for that optimization. All results are contained in parts 1 and 2 of Table 3.3, at the end of the chapter.

$$\begin{tikzcd}
 & \text{base JOINSAT} \arrow[d] & \\
 & \text{Precomputation} \arrow[dl] \arrow[dr] & \\
\text{Collection Match} \arrow[d] & & \text{Active Variables} \arrow[d] \\
\text{Alternate Network} & & \text{Inverse Representation} \arrow[d] \\
 & & \text{Static Analysis}
\end{tikzcd}$$

Figure 3.1: A map of optimizations for JOINSAT. Each optimization is built upon its parent.

3.1 Experimental Problems and Methodology

To compare the various versions of JOINSAT with each other, I select five problems of varying hardness. The last two are the among the hardest (measured by runtime) MACE2 has solved in its many appearances at the CASC ATP Competition [61], which runs a division for satisfiability solving of first order problems. The ground versions of these problems are by no means among the hardest solvable by current SAT solvers, but they are among the hardest solvable with a basic SAT solver unfortified with the crucial optimization of conflict learning, among others. Most problems selected for the competition are mathematical in nature, perhaps reflecting the ATP field's success in proving heretofore open mathematical conjectures. The problem theories are listed in Appendix A.1, but brief descriptions are provided here. All problems have a satisfying model. The full input listing of each problem is included in Appendix A.1; part of the problem output is appended to each input listing. This portion shows which symbols are functions or relations, and also gives the form of the flattened, quantified clauses actually used by JOINSAT to solve the problem.

All experimental runs were on a 1.10 Ghz Athlon PC with 1.5Gb of RAM under the Windows XP OS. If a solver does not return an answer within 10800 seconds (three hours) of CPU time, I consider it to have timed out. The results table includes data on the time taken as well as key performance measures for each solver. The column $\#flips$ measures the total number of flips (assignments of atoms to TRUE or FALSE or back to UNVALUED) made to solve the problem; this column is included for every solver. For MACE2, the number of clause updates made during the run is also reported, while for the JOINSAT solvers the number of entries created and the number of insertions into join indexes are included. For the JOINSAT Collection Match solvers I also report the number of interim entries created.

For the first three problems, derivative clauses are used, but the last two cause so many to be created that they are not used.

3.1.1 Problem Instances

- **cd**. Size of smallest satisfying model: 4. The cd problem is an attempt to show that certain theorems $\mathscr{T}$ of the equivalential calculus are not axioms, by showing that they do not imply another theorem t. This entails finding a model of $\mathscr{T} \cup \{\neg t\}$.
- **o1e1**. Size of smallest satisfying model: 10. Prove that an equation about orthomodular lattices does not hold for ortholattices using one ortholattice.
- **o1e4**. Size of smallest satisfying model: 10. The same idea as o1e1, but with a harder equation.
- **LCL168-1**. Size of smallest satisfying model: 4. Show that XEH is not a single axiom for the R-calculus.
- **BOO061-1**. Size of smallest satisfying model: 6. Show M6D is not an axiom for Boolean algrebra.

3.2 JOINSAT

3.2.1 JOINSAT Results

When evaluated for raw speed, the results for the base JOINSAT algorithm demonstrate a need for optimization. MACE2 is 20-30 times faster than JOINSAT on most problems and the gap in performance appears exponential as problem complexity increases. However, a closer look reveals some hopeful statistics. From a theoretical perspective, MACE2's runtime is dominated by the total number of clause updates it performs, while JOINSAT's runtime is dominated by the total number of entries it creates and the total number of insertions of entries into join indexes. Since most, but not all entries created are inserted into one or two indexes, there is a great deal of overlap in these two figures, so I refer to the number of insertions as a better indicator of JOINSAT's complexity.

Now, in at least some of the problems, JOINSAT makes fewer inserts than MACE2 makes updates, indicating that the JOINSAT algorithm is potentially more efficient in some cases. The overall exponential lag certainly exists and probably represents a real complexity difference between the two methods. We saw in section 2.3 that join network method complexity can be greater than ground instantiation method complexity for problems having large models and few join variables. This disparity can be even more pronounced when the ground instantiation method's complexity is reduced by means of clause splitting. In later versions of JOINSAT (especially the Static Analysis version) I reduce its complexity enough to make the algorithm competitive even with an optimized ground instantiation approach; in section 4.2.1 this comparative complexity is discussed at more length.

JOINSAT is also hampered by enormous overhead costs associated with creating and detroying complex structures representing entries and atoms and bindsets. Furthermore, the original version of JOINSAT incorporates various sources of unnecessary matching, such as extending the join network to detect assignment inconsistency (by matching upon all n literal nodes, instead of just $n-1$). This extraneous matching vastly inflates its #entries and #insertions numbers, so in actuality JOINSAT's theoretic measures are better than are shown. Later versions cut out these sources of inefficency.

3.3 Optimization 1: Precomputation

I noted an important weakness of the base JOINSAT algorithm in section 2.6: its reliance upon dynamic memory operations, including the creation of structured objects like atoms and entries, and insertion into indices implemented as dynamic hash tables. Test runs show that over 85% of runtime is devoted to these operations. This is in stark contrast to MACE2, in which almost all data structures are allocated at runtime, and dynamic memory computations are rare.

Accordingly, my first effort to optimize JOINSAT involves the elimination of these dynamic memory operations. This is achieved this using precomputation methods, in three steps. First, a mapping is constructed of every entry and atom to a distinct integer. Then, for each dynamic memory computation upon entries and atoms, an n-dimensional lookup table is constructed that represents every possible instance of the computation. In each table, a computation instance's n inputs are represented by n integer indices into the table, and the computation instance's output by the integer stored in that cell of the table. Finally, the original computation code is replaced with a table lookup. If all computations throughout the algorithm upon atoms and entries are replaced with table lookups, then the outputs for one computation are in exactly the integer format needed for use as inputs to the next computation.

First the atoms are mapped. For every predicate, a mapping F is constructed that associates each possible ground atom instantiating that predicate with a distinct integer value, e.g. for predicate p and domain size 3, $F(p(1,1)) = 1$, $F(p(1,2)) = 2$, and so on until $F(p(3,3)) = 9$.

Next, the same function F is constructed for all possible bindsets, although I represent these as ordered vectors of their bindings, ignoring the variables. For example, the bindset $\langle x = 1, y = 2\rangle$ is represented as the vector $\langle 1,2\rangle$. the first step is to determine the maximum number of bindings n that will be found in any bindset computed by the join networks constructed for the problem. For example, in the join network in Figure 2.5, the largest bindset is found in entries computed for join J_{23}, and the number of bindings in this bindset is 5: one binding for each of p_1, l_1, t_1, t_2 and l_2. Then, for each $k \leq n$, each k-ary vector of domain values is mapped to a distinct integer. So, for example, if $d = 3$, $F(\langle 1\rangle) = 1$, $F(\langle 2\rangle) = 2$, and $F(\langle 3\rangle) = 3$; the sequence restarts when the size increases, so $F(\langle 1,1\rangle) = 1, F(\langle 1,2\rangle) = 2, \ldots, F(\langle 3,3\rangle) = 9$, and so on.

Now that the mapping function is constructed, various computations can be precomputed and cached in lookup tables. For example, the process of indexing and matching inputs (JOINSAT-FLIP, in Figure 2.13, lines 14-15) can be precomputed. Instead of using a dynamic hash table to store entries, an array is made with one cell for every possible bindset upon the joined variables. Instead of using a hash function to compute an index key, the index keys for every possible input are stored in a lookup table. The index key, is of course, an index into the array of index cells. So, after looking up the proper cell, the integer representing the input entry is inserted into that cell.

This process is illustrated in Figure 3.2: the figure shows a precomputed index mechanism for join J_{23} from Figure 2.5. This join matches entries having the same binding for variable x. Since $d = 3$, there can only be three possible bindings, so J_{23}'s index array has three cells. Since J_{23}'s left entries are bindsets upon v, w and x, we construct a lookup table of size $d^3 = 27$; the lookup table for right entries has size $d^2 = 9$.

Suppose we have entry $e_4 : \langle v = 1, w = 2, x = 3\rangle$ and entry $e_3 : \langle x = 3, y = 2\rangle$. In vector form, the two inputs are $\langle 1,2,3\rangle$ and $\langle 3,2\rangle$. Since $d = 3$, we have $F(\langle 1,2,3\rangle) = 6$, $F(\langle 3,2\rangle) = 8$. So, when entry e_4 is passed as a left input to J_{23}, cell 6 is accessed in the left lookup table. This cell contains the key value 3, so the entry (i.e. its integer, 6) is inserted into the third index cell. A similar process is used to index the right entry e_3, inserting its integer value (8) into the third cell.

Another costly operation that can be partially precomputed is the creation of new match entries (JOINSAT-FLIP, in Figure 2.13, lines 16-17). When two join inputs match, they cause a new entry to be created; this process can be seen as a mapping of two bindsets (the inputs) to a third (the match). This can be modeled by constructing a lookup table for a particular join containing the results of all successful matches at that join. This lookup table is a two-dimensional array of size $d^m * d^n$, where m and n are the bindset sizes of the join's left and right inputs, respectively. The range of values contained in the lookup table should be d^o, where o is the size of the bindset in succesful match entries created by the join. Note that for a given join, these sizes are the same for every match, e.g. a join will never receive inputs from its left parent of different sizes.

This operation is portrayed in Figure 3.3, again using join J_{23}. This join receives a left input from join J_{22} whose bindset is (always) of size 3 containing variables v, w, and x ; its right input from literal node L_{13} has a bindset of size 2 containing variables x and y. So, the match lookup table constructed for J_{23} has dimensions of d^3 and d^2; its total size is therefore $d^{3+2} = d^5$. When these inputs match, the join creates a new entry with a bindset of size 4, containing variables v, w, x and y. Therefore, the values contained in the lookup table range between 1 and d^4. This lookup table is populated by iterating through every possible match of a 3-ary bindset with a 2-ary bindset and computing the output bindset; then the three values are converted to integers. For example, entry e_4 : $\langle v = 1, w = 2, x = 3\rangle$ matches with bindset e_3 : $\langle x = 3, y = 2\rangle$, causing the creation of bindset e_5 : $\langle v = 1, w = 2, x = 3, y = 2\rangle$. In vector form, the two inputs are $\langle 1,2,3\rangle$ and $\langle 3,2\rangle$; the output is $\langle 1,2,3,2\rangle$. Assuming $d = 3$, we have $F(\langle 1,2,3\rangle) = 6$, $F(\langle 3,2\rangle) = 8$, and $F(\langle 1,2,3,2\rangle) = 17$; accordingly, the value 17 is assigned to Lookup$(6,8)$.

Other particularly expensive operations that are converted to table lookups include: matching a ground literal against a clause literal (JOINSAT-FLIP, in Figure 2.13, lines 6-7) and the process of incrementing bindings and creating *least_unvalued_target* atoms (INCREMENT_LEAST_UNVALUED_TARGET, in Figure 2.14, lines 3-14 and 17-23). The latter is particularly interesting because all the code for incrementing bindsets is eliminated. Instead, for a given join, each possible entry's target vector is explicitly computed and stored as a vector of integers. So, to increment *least_unvalued_target* the algorithm simply looks at the next value in the vector.

A problem that arises in some of these lookup computations is that the lookup table becomes large enough to degrade performance. For example, as stated above, the matching process requires a lookup table of size $d^m * d^n$, where m and n are the bindset sizes of the join's left and right inputs, respectively. This can easily create tables with hundreds of thousands or even millions of entries. One solution is to avoid redundant information whenever possible. Consider the example above, in which the left input and the right input both contain the variable x. To reduce the size of the lookup table an intermediate table is introduced that compresses the right input, which contains the variables x and y, to a new bindset containing only y. This allows the use of a smaller lookup table whose left input has variables v, w, and x, and whose right input has variable y. The total size of this improved lookup table is therefore d^4, instead of d^5.

One important point to note is that by shifting to this ground representation, I make JOINSAT vulnerable to some, but not all, of the weaknesses of MACE2's entirely ground representation. An exponential amount of lookup table space is allocated at runtime, so if the problem is so large that simply allocating space for it at runtime is prohibitively costly, then JOINSAT's performance will suffer, just as MACE2's does.

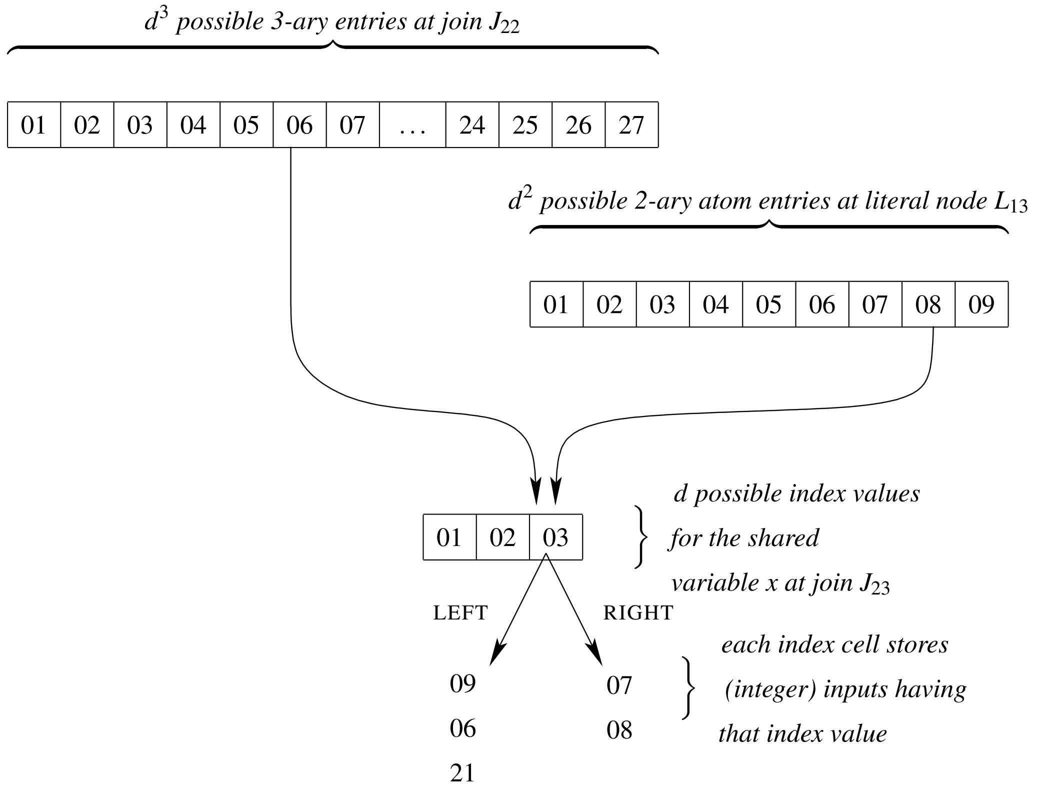

d^3 possible 3-ary entries at join J_{22}
01 02 03 04 05 06 07 ... 24 25 26 27
d^2 possible 2-ary atom entries at literal node L_{13}
01 02 03 04 05 06 07 08 09
01 02 03
d possible index values for the shared variable x at join J_{23}
LEFT
RIGHT
09
06
21
07
08
each index cell stores (integer) inputs having that index value

Figure 3.2: An example of join indexing using precomputation.

d^3 possible 3-ary entries from join J_{22}

d^2 possible 2-ary atom entries from literal node L_{13}

	01	02	03	04	05	06	07	…	24	25	26	27
01												
02												
03												
04												
05												
06												
07												
08						17						
09												

Figure 3.3: An example of match entry creation using precomputation. Each cell stores the (integer) match entry created from the two inputs.

However, the complexity of JOINSAT's operations remains unchanged: the computation described in section 2.2 and Figure 2.3, for example, will still only require 2 matches. The rationale for this shift to an exponential representation is that by removing the very large overhead of dynamic memory operations, it allows a more accurate contrast between the performance of MACE2 and JOINSAT. In future work it may be possible to optimize the dynamic memory operations of the base JOINSAT algorithm in some fashion that avoids exponential runtime memory allocation.

Another limitation of the current implementation is that not all dynamic memory access is eliminated. Join indexes are still implemented as linked lists inside each join cell, as shown in Figure 3.2.

3.3.1 Precomputation Results

The precomputation version greatly improves runtime over the base version of JOINSAT. Perhaps a 2x speedup is due to the elimination of unnecessary matching in the precomputation version. The rest arises from the greatly streamlined representation and the avoidance of dynamic memory access.

Because they do not reflect useless matching, the numbers of entries created and insertions made for this version are a better measure of the theoretical complexity of the base JOINSAT algorithm. Note that in all but the hardest problem, JOINSAT is making half or less as many insertions as MACE2 is making clause updates. Part of this is because of MACE2's clause splitting preprocessing technique, explained in section 1.7.2.7. The clause splits increase the number of clauses in the problem, which increases the size of the search space. Though the resulting search space increase seems to be linear instead of exponential, it does usually require MACE2 to search through four to ten times as many nodes as JOINSAT.

However, it must be admitted that in the BOO061 problem MACE2 has better theoretical numbers in addition to much greater speed. This shows that there are indeed some situations in which join network matching is intrinsically more complex than MACE2's method of ground clause updates. One reason for this is that BOO061's clauses are very long and contain a very large number of variables. Because JOINSAT, unlike MACE2, does not split these clauses and thereby reduce the number of clause variables, JOINSAT can suffer from its own version of blowup. This is because the precomputed lookup tables are exponential in the number of variables used to create them. We do show later that JOINSAT's theoretical numbers can be improved.

3.4 Optimization 2: Collection Match

While the precomputation optimization greatly mitigated the problem of dynamic memory usage, it does not address the large table problem found in a number of the problem instances. Joins upon large tables produce a great number of entries that require further matching, whether that matching is represented in precomputed form or not. Fortunately, many of these entries are superfluous, and can be winnowed away. My second optimization is an effort to use this fact to reduce the number of matches arising from large tables. This optimization is as extensive as was the first, requiring new data structures and the rewritng of large parts of the JOINSAT-FLIP() algorithm. The object is to alter the network to use Collection Match, an efficient matching algorithm used by production systems. The next section, 3.4.1, gives a brief account of the original Collection Match algorithm, and section 3.4.2 describes my adaptation of it to join networks.

3.4.1 Collection Match in Production Systems

Collection Match [2] involves a new representation for match tuples computed by a production network. It is based upon the insight that the normal join process computes sets of matches or tuples using a cross-product computation. The algorithm refrains from instantiating this cross product, i.e. it does not compute all possible pairings/matches between elements of the two sets. Instead, the cross-product is represented it as a pair of sets, and individual pairings are only computed lazily, upon demand. I illustrate the basic idea using join networks.

Consider the simple join network in Figure 3.4. Let us assume that the entries in the

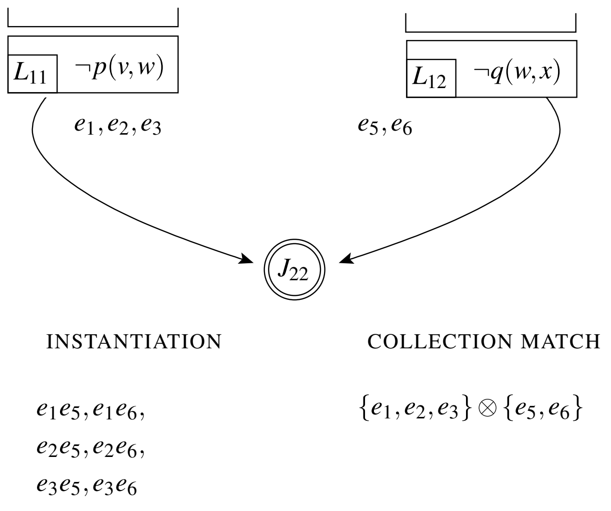


Figure 3.4: The idea behind Collection Match.

figure are defined at their associated literal nodes, ignoring for the moment the ground literals that define them. Assume further that each left input entry matches with each right input entry. On the lower left, the matches that would be made using the standard joining algorithm are shown; each of these matches would produce a new entry at the join. On the lower right a collection match representation is shown, in which the individual matches are not computed. Instead, the matches are represented as the cross-product of the two input sets. Clearly, as the number of inputs increase, the collection match representation will be smaller by a factor of $n^2/2n$, where n is the number of inputs on each side.

The original Collection Match algorithm's basic datum was a token: a vector of tuple sets $\langle s_1, s_2, \ldots, s_n \rangle$ that represented the cross product of those sets, i.e. $s_1 \otimes s_2 \otimes \cdots \otimes s_n$. Tokens were passed down the production memories instead of the tuples portrayed in section 2.1. In a production with multiple complex preconditions, the savings achieved by this representation can be multiplicative, as multiple successive cross-product instantiations are avoided. So, for example, if all the n^2 matches from one join were successfully matched with n more inputs at a second join, the total number of matches would

Literal Node	Entries
$L_{11} : p(v,w)$	$e_1 : \{1,1\}, e_2 : \{2,1\}, e_3 : \{3,1\}, e_4 : \{2,2\}$
$L_{12} : q(w,x)$	$e_5 : \{1,1\}, e_6 : \{1,2\}, e_7 : \{2,1\}, e_8 : \{3,1\}$
$L_{13} : r(x,y)$	$e_9 : \{1,1\}, e_{10} : \{1,2\}, e_{11} : \{3,1\}, e_{12} : \{4,2\}$

Table 3.1: Entries defined at each node of Figure 2.3.

be n^3. The Collection Match representation would meanwhile become a cross-product of three sets, now requiring space and time $3n$. So, over an entire network, the savings can potentially be exponential.

3.4.2 Adapting Collection Match to Join Networks

In this section my adaptation of Collection Match to a join network context is discussed; I believe this adaptation, called CM-JOINSAT, is more efficient than the original Collection Match algorithm.

The adaptation is illustrated using the example problem and join network from Figure 2.3, but this time with a different assignment. Ground literals are ignored for the moment: instead, a given set of entries defined at each literal node is assumed. Let us assume the sets of entries shown in Table 3.1 are defined at their corresponding literal nodes, e.g. entry e_1 is defined at literal node L_{11} by some unnamed literal. For brevity, the entries are shown as vectors of values, so, e.g., e_1 is actually the bindset $\{v = 1, w = 1\}$.

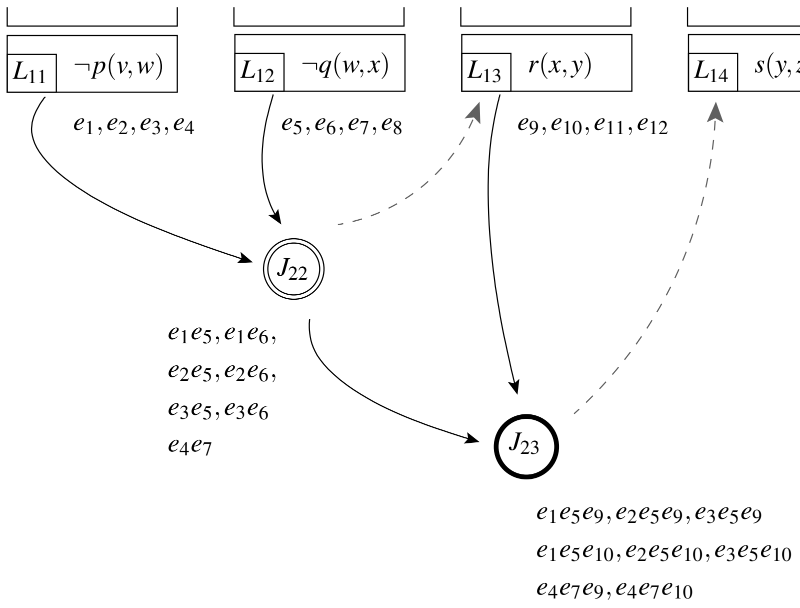
Figure 3.5 (a) shows how these entries would match in a standard join network; 3.5 (b) shows the corresponding matching in a CM join network. In the first diagram, match entries are portrayed as simple conjunctions of their inputs, e.g. e_1e_5 stands for the new entry that is created when e_1 and e_5 match.

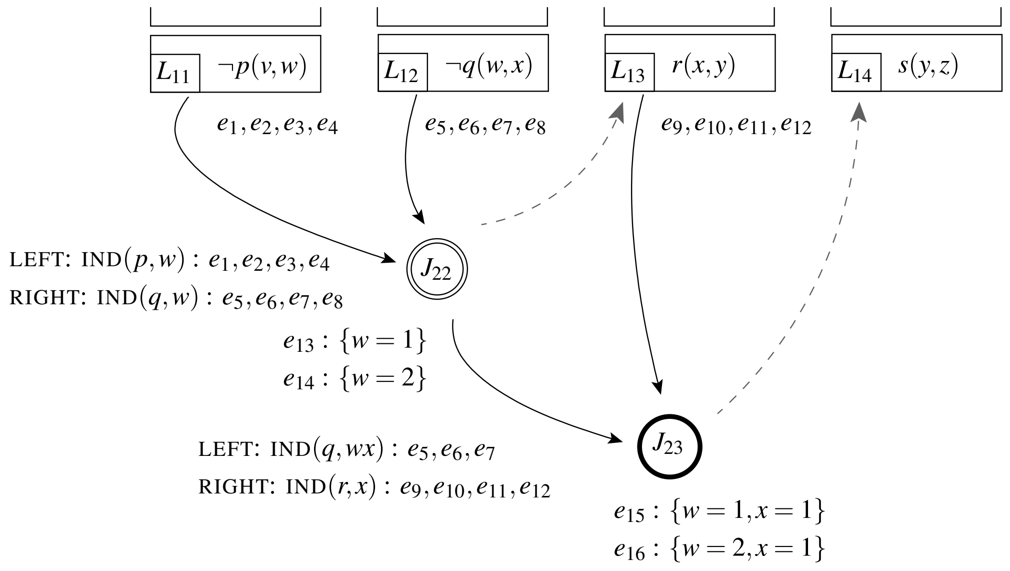
A core difference between the two networks is how they define the match entries created by join networks. In the standard network, a match entry contains a variable binding for every variable contained in any ancestor of the join. For example, e_1e_5 contains bindings for the variables v and w, which are found in literal node L_{11}, and also contains a binding for the variable x, found in literal node L_{12} (along with another instance of w). In contrast, a match entry in the lower network contains bindings only for variables that have been joined, either at the current join or at some ancestor join. So, match entries in join J_{22} contain only the variable w joined upon at J_{22}. Match entries in join J_{23} contain w, as a formerly joined variable, and also x, which is joined at J_{23}.

CM match entries have different variables because they have a different semantics. In a standard network, match entries are simply bindsets that falsify portions of the network.. But in a CM join network, match entries are stand-ins for *cross-products of literal node entries* that falsify portions of the network. For example, consider the entry $e_{13} : \langle w = 1 \rangle$ at join J_{22}. This entry is a stand-in for the cross-product $\{e_1, e_2, e_3\} \otimes \{e_5, e_6\}$ composed of those left and right inputs to J_{22} having the matching binding $\langle w = 1 \rangle$.

In keeping with this semantics, CM match entries are not matched together at joins: instead, the cross-products they represent are matched. For example, at the next join, J_{23}, entry e_{13} (representing the cross-product $\{e_1, e_2, e_3\} \otimes \{e_5, e_6\}$) is matched together with the singleton cross-product $\{e_9, e_{10}, e_{11}, e_{12}\}$ consisting of the literal entries in literal node L_{13}. The result is a new cross-product $\{e_1, e_2, e_3\} \otimes \{e_5\} \otimes \{e_9, e_{10}\}$, which is composed of precisely those literal entries having the bindings $\{w = 1, x = 1\}$, and which is represented by $e_{15} : \{w = 1, x = 1\}$ at join J_{23}.

Stand-in entries are a parsimonious way of representing cross-products, but they do not contain all the information needed to match cross-products together. For matching, the cross-product is reconstructed from its stand-in by fetching all the input entries again. To do this efficiently, successive multiple indexes for each literal must be maintained. For example, to build the cross-product associated with e_{13}, the literal entries from literal $\neg p(v, w)$ (i.e. literal node L_{11}) having $w = 1$ must be obtained. However,





L_{11} $\neg p(v,w)$
L_{12} $\neg q(w,x)$
L_{13} $r(x,y)$
L_{14} $s(y,z)$
e_1, e_2, e_3, e_4
e_5, e_6, e_7, e_8
$e_9, e_{10}, e_{11}, e_{12}$
LEFT: IND(p,w) : e_1, e_2, e_3, e_4
RIGHT: IND(q,w) : e_5, e_6, e_7, e_8
J_{22}
$e_{13} : \{w = 1\}$
$e_{14} : \{w = 2\}$
LEFT: IND(q,wx) : e_5, e_6, e_7
RIGHT: IND(r,x) : $e_9, e_{10}, e_{11}, e_{12}$
J_{23}
$e_{15} : \{w = 1, x = 1\}$
$e_{16} : \{w = 2, x = 1\}$

Figure 3.5: An example of Collection Match.

to build the cross product associated with e_{15}, entries having both $w = 1$ and $x = 1$ must be obtained from L_{11}. For greater efficiency, new indices are built at the joins for which they are needed: for example, in Figure 3.5 (b), $\textsc{ind}(p, w)$, at join J_{22}, denotes an index of the entries from literal $\neg p(v, w)$, all indexed by their binding for variable w. A new IND is built at a join for a particular literal each time some variable from that literal is joined for the first time. Successive indexes also contain fewer of the original set of literal entries, because literal entries are only passed to index $n+1$ if some stand-in entry has the right bindings to pull them from from index n.

To explain the incremental matching process in greater detail, I walk an example through the CMJOINSAT code. Aside from the code setting up the data structures mentioned above, the changes to the JOINSAT algorithm needed to implement Collection Match all concern the JOINSAT-FLIP procedure. Therefore, an alternative procedure, CM-JOINSAT-FLIP, shown in Figure 3.7, is introduced. The example begins in the state shown by Figure 3.5 (b). Then atom $p(4, 1)$ is flipped to true, resulting in the network state shown in Figure 3.6. I now detail how these state changes are computed.

Lines 1-12 of CM-JOINSAT-FLIP are identical to those in JOINSAT-FLIP; I therefore skip them, except to note that the literal entry produced by the matching process between $q(4, 1)$ and literal node L_{11} in lines 6-9 is $e_{17} : \{4, 1\}$. After *join* is given the value J_{22}, line 14 calls MAKE-CROSS-PRODUCT(e_{17}, e_{17}, J_{22}), pictured in Figure 3.8. This procedure reconstructs a cross-product defined by *entry* and instantiates it into individual entries/bindsets that can be matched at the join.

The first task of MAKE-CROSS-PRODUCT, implemented in lines 2-9, is to collect (in *entry_sets*) sets of literal entries that match *entry* from indices in ancestors of *join*; together, these sets comprise the cross-product to be instantiated and returned. The following rule determines whether a set is collected for a given literal: if any variable in the literal is joined upon at *join*, then a set must be collected for it. The *join* contains a set *inds* of pointers to all the ancestor indices from which a literal set is to be collected; *inds* is divided according to the parents of *join*. In this case, J_{22} has one index in $inds[L_{11}]$: $L_{11}.entries$.

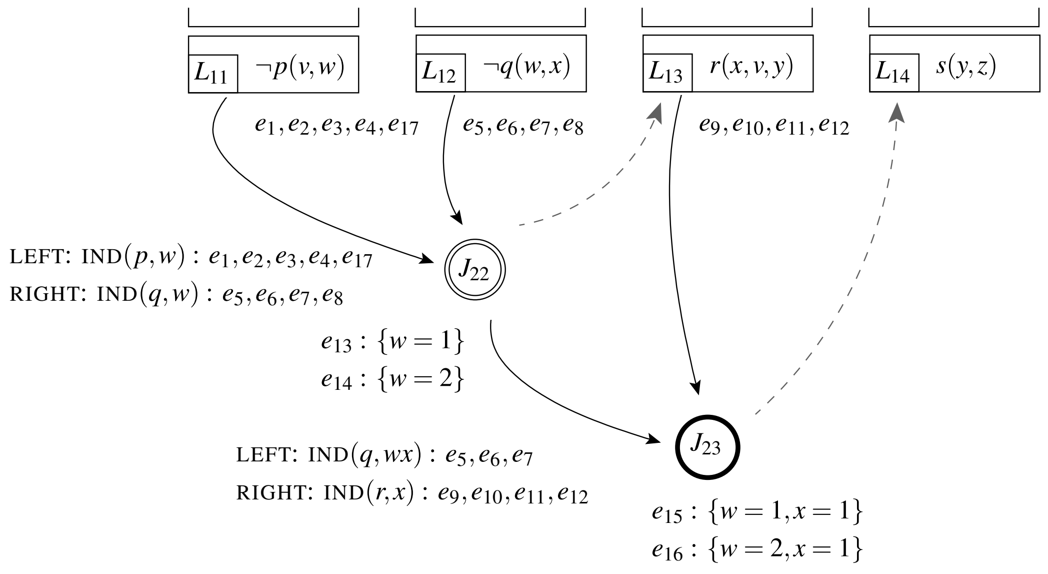


Figure 3.6: The Collection Match example extended to state two.

Cases govern the set to be collected. In the case (line 2-3) that *ind* indexes *literal_entry*'s literal node, as occurs with e_{17} and L_{11}.*entries*, the singleton set $\{literal\_entry\}$ is pushed onto *entry_sets*. In the case (line 4-5) that *ind* is the *entries* table for some literal node, that entire *entries* set is pushed onto *entry_sets*. Otherwise (lines 6-8), *entry* is used as a key into *ind*, and the set of all literal entries retrieved is pushed onto *entry_sets*.

So, in this case, after all sets are collected, *entry_sets* is $\{\{e_{17}\}\}$. Now, the eventual point of collecting all these sets is to be construct entries/bindsets having precisely the variables required by the current join. Therefore, every collected set must be re-indexed using these join variables (lines 9-12). For each collected set, *join* contains a new index *new_inds*[*ind*], and every literal entry in the set is inserted here using any of its (previously or currently) joined variables. In the example, e_{17} is inserted into IND(p,w) using the join variable *w*.

In lines 13-17 of MAKE-CROSS-PRODUCT, the sets in *entry_sets* are crossed together. Line 13 calls get_cross_products(), which returns the instantiation of the cross-

```
procedure CM-JOINSAT-FLIP($S, atom, val$)
 1: push($val, atom.value$)
 2: for all $entry \in atom.update\_entries$ do
 3:     INCREMENT-LEAST-UNVALUED-TARGET($entry, S$)
 4: $atom.update\_entries$ := $\phi$
 5: for all $lnode \in S.literal\_nodes$[negate($val$)][$atom.predicate$] do
 6:     if $bs$ := unify($atom.args, lnode.args$) then
 7:         $literal\_entry$ := make_entry($bindset, lnode$)
 8:         push($literal\_entry, entries$)
 9:     while $entries$ is not empty do
10:         $entry$ := pop($entries$)
11:         $node$:= $entry.node$
12:         push($entry, lnode.entries$)
13:         for all $join \in node.join\_nodes$ do
14:             $products$ := MAKE-CROSS-PRODUCT($entry, literal\_entry, join$)
15:             for all $product \in products$ do
16:                 $index\_bs$ := filter_bindset($product.bs, join.joinvars$)
17:                 insert($product, join.index[index\_bs][node]$)
18:                 for all $product2 \in join.index[index\_bs][join.otherparent[node]]$ do
19:                     $bs2$ := $product.bs \cup product2.bs$
20:                     $entry2$ := make_entry($bs2, join$)
21:                     push($entry2, entries$)
22:         if $node \in S.propagation\_joins$or $node \in S.selection\_joins$ then
23:             INCREMENT-LEAST-UNVALUED-TARGET($entry, S$)
```

Figure 3.7: CM-JOINSAT-FLIP($S, atom, val$).

product of *entry_sets*: the set $\Sigma = \{e_1, e_2, \dots e_n : e_i \in entry\_sets[i]\}$. In this case, the computation is quite simple and get_cross_products() returns $\Sigma = \{\{e_{17}\}\}$.

The rest of the procedure operates individually on each entry set $\varepsilon \in \Sigma$. For each ε, all bindings in its entries are consolidated (line 14) and any variables not joined at *join* are filtered away (line 15). Then, a new *product* entry is created using these bindings and the original $entry's$ bindings (line 16), and added to the set *products* (line 17), which is later returned (line 18). So, in the example the one resulting *product* entry is $p_1 : \{w = 1\}$.

Returning to CM-JOINSAT-Flip, in lines 16-21 each *product* is matched upon in a manner identical to that in which *entry* is matched upon in JOINSAT-FLIP lines 14-19. In lines 16-17 of CM-JOINSAT-FLIP, a join key is obtained from *product* and used to insert it into *join.index*. Inputs from the opposite parent having the same join key are matched with *product* in line 18, and new match entries are created in lines 19-21.

There are two differences that are not reflected in the code. The first is that new match entries will not contain all the variables in ancestor literals, but rather only variables previously or currently joined upon. The second concerns line 17, when *product* is inserted into *join.index*. In contrast to JOINSAT-FLIP, in which inserted entries are unique and so are never duplicates of earlier insertions, in CM-JOINSAT-Flip duplicates are a commonplace, because *product* and *entry* are used to stand for sets of literal entries. So, in this case, product p_2 is a duplicate of an earlier product inserted when the literal entry e_1 (which has the same binding for w as does e_{17}) was added. Though it is not reflected in the code, insertions throughout the code are checked for duplication and not performed in the case of duplicates: the duplicates are eventually discarded after the code is done with them.

Now, in the example, *product* matches with an identical product input from L_{12}; this product must have the exact same bindset as does *product*, because at this stage bindsets consist solely of current join variables. The new match entry (call it e) must therefore be $e : \{w = 1\}$. This entry is identical to the entry e_{13}, so we may continue the example acting as if e_{13} had been created again. However, it is worth noting that e_{13} was, in

```
procedure MAKE-CROSS-PRODUCT($entry, literal\_entry, join$)
1:  for all $ind \in join.inds[entry.node]$ do
2:      if $literal\_entry.node = ind.literal\_node$ then
3:          push($\langle literal\_entry \rangle, entry\_sets$
4:      else if $ind.node \in S.literal\_nodes$ then
5:          push($ind.node.entries, entry\_sets$)
6:      else
7:          $ind\_bs$ := bindset_filter($entry.bs, ind.vars$)
8:          push($ind[ind\_bs], entry\_sets$)
9:      $new\_ind$ := $join.new\_inds[ind]$
10:     for all $entry2 \in entry\_sets.last$ do
11:         $ind\_bs2$ := bindset_filter($entry2.bs, new\_ind.vars$)
12:         insert($entry2, new\_ind[ind\_bs2]$)
13: for all $entry\_set \in$get_cross_products($entry\_sets$) do
14:     $union\_bs := \bigcup_{entry3 \in entry\_set} entry3.bs$
15:     $joinvar\_bs$ := filter_bindset($union\_bs, join.joinvars$)
16:     $product$ := make_entry($entry.bs \cup joinvar\_bs, join$)
17:     push($product, products$)
18: return $products$
```

Figure 3.8: MAKE-CROSS-PRODUCT($entry, literal\_entry, join$).

the previous state, a stand-in for the cross-product $\{e_1, e_2, e_3\} \otimes \{e_5, e_6\}$, i.e. all literal nodes from L_{11} and L_{12} having $w = 1$. Now that e_{17} has been inserted, e_{13} becomes the stand-in for the cross-product $\{e_1, e_2, e_3, e_{17}\} \otimes \{e_5, e_6\}$, yet this update did not at any time require any matching between individual elements of the cross-product.

After e_{13} is popped from *entries* in lines 10-12, MAKE-CROSS-PRODUCT is called again with $entry = e_{13}$, $literal\_entry = e_{17}$, and $join = J_{23}$. In this case, $J_{23}.inds[J_{22}]$ consists of $\text{IND}(q, w)$. I should note here that *inds* does not include any index for literal p, despite the fact that any cross products matched and created at join J_{23} will indeed contains sets of literal entries from L_{11}. This is because these entries contain no information (i.e. join variables) needed for the current join. We may therefore perform matching without consulting them, confident that any new match entry contains some binding for w that will allow us to eventually retrieve the appropriate values from $\text{IND}(p, w)$.

For *ind* of $\text{IND}(q, w)$, neither of the case in lines 2 or 4 apply. Therefore, in lines 7-9, e_{13} is used as a key to extract the relevant literal entries from $\text{IND}(q, w)$. In line 7, the key bindset $\{w = 1\}$ is obtained from e_{13} ($\text{IND}(q, w)$ knows it is indexed upon w). In line 8 the set of those entries in $\text{IND}(q, w)$ having $w = 1$ is pushed onto *entry_sets*: this set is $\{e_5, e_6\}$. These entries are inserted using their bindings for w and x into the new index $\text{IND}(q, wx)$ in lines 9-12.

Again the process in line 13 of instantiating the cross product of *entry_sets* is comparatively simple, producing the set $\{\{e_5\}, \{e_6\}\}$. Product entries containing past and present join variables are made and returned for each of $\{e_5\}$ and $\{e_6\}$: let us call these $p_1 : \{w = 1, x = 1\}$ and $p_2 : \{w = 1, x = 2\}$ respectively. We return again to CM-JOINSAT-Flip, line 15 . These returned products are duplicates of products formed when e_{13} was previously inserted, so the insertions in line 17 do not take place. However, each product is matched against product inputs coming from L_{13}. Product p_1 matches with products defined by the literal entries $e_9 : \{x = 1, y = 1\}$ and $e_{10} \{x = 1, y = 2\}$: these matches form duplicates of entry e_{15}. Product p_2's binding for x prevents it from matching successfully.

Again taking stock of these computations, we note that entry e_{15}, which hitherto represented the cross product $\{e_1, e_2, e_3\} \otimes \{e_5\} \otimes \{e_9, e_{10}\}$, has been updated to now represent the cross product $\{e_1, e_2, e_3, e_{17}\} \otimes \{e_5\} \otimes \{e_9, e_{10}\}$, without individually computing either of the new implicit bindings $e_{17}e_5e_9$ or $e_{17}e_5e_{10}$. More generally, as the number of literal entries grow large, CM-JOINSAT offers the promise of maintaining a compact representation even in the presence of many matches. This promise is not fully realized in my work: I discuss why in the next section..

3.4.3 Collection Match Results

The results for JOINSAT-CM are disappointing. The overhead required to maintain the new indexes stationed along the join network seems to outweigh the theoretical reduction in total matching. Part of this reduction is illusory, because Collection Match sometimes just defers matching instead of eliminating it. For example, if we make a cross product at a join, and then totally instantiate it at the next, we are effectively doing the matching at the second join that we avoided in the first. To be truly effective, Collection Match must instantiate only portions of the cross product at each join, pruning away selected sets while avoiding full instantiations. This is not always possible in the problems used for testing. One bright side is that Collection Match's theoretical numbers are better than those of every prior versions: it requires substantially fewer matches and inserts, and even the more dominant interim entry creation is less complex than prior versions. However, its massive overhead makes it infeasible at present.

3.5 Optimization 3: Alternate Networks

The results thus far show that the problem of large tables discussed in section 2.6.0.3 is serious and not easily addressed. Because problems often contain non-negated functional literals, often the number of matches computed by the rightmost joins in JOINSAT's network (where these literals are joined) are orders of magnitude greater than in the leftmost joins. This section discusses another attempt to mitigate this problem by designing a new join network in which the order in which literals are joined is carefully reworked to delay the joining of non-negated functional literals.

Any attempt at an alternate join network needs to obey the following requirements. First, it must perform the expected subsearch functions. The most challenging of these, as discussed in section 2.4.2, remains unit propagation. This is because propagation must be computed for every literal node and because propagation requires a network that excludes the propagated literal node.

Another requirement for an alternate network is that it avoid joining upon large tables as long as possible. Since non-negated nodes are large tables, and since I arrange literal nodes so that non-negated nodes are on the right, any network minimizing matching will presumably have most join paths flowing left to right so the matching of large tables may be delayed. My personal experiments show that by joining a large table after even one less expensive join, the costs can be cut dramatically. However, an important component of the network introduced in section 2.4.2.1 was the network of joins in level 3 running right to left. The repeated matching of rightward-flowing level 2 joins with leftward-flowing level 3 joins allowed us to compress the unit propagation network from n^2 joins down to n joins. If the leftward-flowing level 3 is eliminated, it is not immediately clear whether the network can remain small and still compute propagation.

Figure 3.9 shows an alternate network that overcomes these problems as well as might be hoped. Its first three levels flow rightward, but are distinct because each one joins a different pair from the leftmost three nodes. In doing so, each level skips a different node, and ultimately computes propagation for that node. The fourth level flows leftward, allowing it to join repeatedly with level 3 (at level 5) and therefore computes propagation for the middle literal nodes efficiently in the manner of the old network. However, the fourth level does not begin with a dangerously expensive join upon the rightmost two nodes: instead, the rightmost node is joined with a join node from the first level, hopefully mitigating this most costly of joins. Indeed, minimizing the cost of the join at $J_{4,n}$ shapes the structure of the rest of the network, in the following way. If we join J_{11} with $J_{4,n}$, then level 4 cannot help compute propagation for literal nodes L_{11} and L_{12}: a propagation network for a node cannot have it as an ancestor. Therefore,

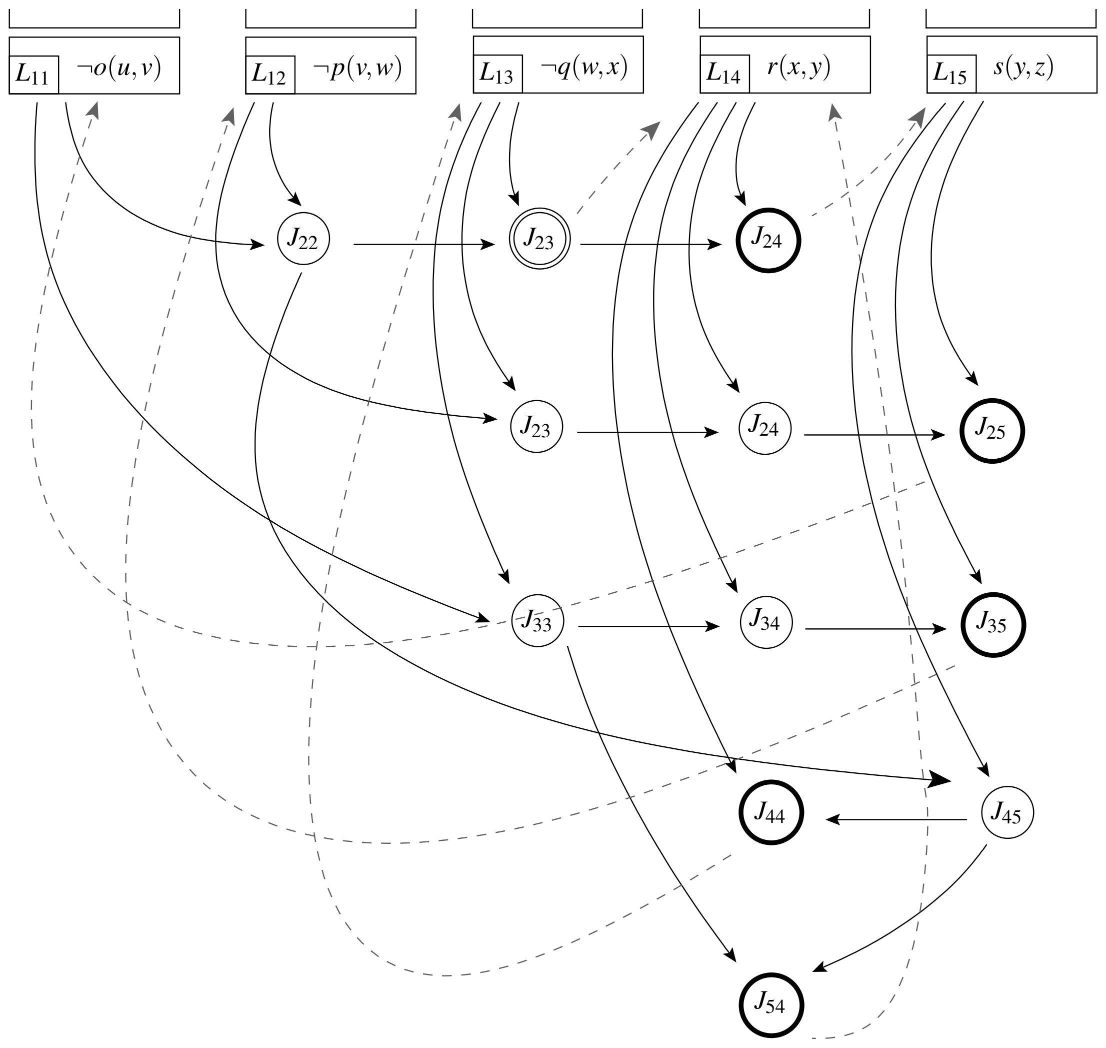


Figure 3.9: An alternate join network for subsearch functions.

levels 2 and 3 are needed to compute propagation for L_{11} and L_{12}, resepctively.

This alternative network meets the requirements laid out above, but at a cost. First, it has five levels instead of three. And second, where no literal node in the old network has more than two child joins, several literal nodes in the new network have three or four child joins. This additional overhead can sometimes outweigh the theoretical gains achieved by avoiding the early join of large tables.

3.5.1 Alternate Networks Results

The results for the alternate network are also disappointing. The network does not improve upon any of the numbers of its predecessor, JOINSAT-Collection Match. A major reason for this is the importance of insertions made by literal nodes. Because the focus is to reduce the amount of matches made at join nodes, it is easy to overlook that almost 30% of all insertions made in a typical problem are join insertions of entries passed by literal nodes, *before* any matching has taken place. Therefore, a network that greatly increases each literal node's number of child joins cannot increase overall efficiency. A better way must be found to counteract the deleterious effects of large table literal nodes.

3.6 Optimization 4: Active Variables

In this section, I describe a different approach to mitigating the large table problem: an optimization that eliminates match entries that are redundant, in the sense that the bindings that distinguish them from other entries are unused by future joins. In the process of eliminating these duplicates, the entries that remain are compressed by discarding these unused bindings. Whereas the compression described in connection with precomputation (in section 3.3) just makes lookup tables smaller, the compression descirbed here actually reduces the number of total matches/entries made by the algorithm, and so deserves its own section.

Consider join J_{22} of Figure 2.6 and an arbitrary entry/bindset e this join outputs. The variables in the bindset are v, w, and x. Now, the variable binding for x is actually used in further computations. For example, join J_{23} takes e as its left input and uses $e's$ binding for x for matching. Also, since J_{23} is a selection node, entry e's binding for x partially determines its vector of *least_unvalued_target*'s, thereby determining which literals might be flipped next. Since x is potentially used in future computations, I call it an *active variable*. However, e's other variables v and w are not active variables: they are not used in any further matching or target determination. To confirm this, we may appraise some entry e' whose binding for x is identical to e's, but whose bindings for v and w are different. This entry e' will have precisely the same target vector as does e, and it will match precisely the same right inputs in join J_{23}. Which, indeed, is the point: e and e' cause work to be duplicated unnecessarily.

To eliminate this isomorphism, first the set of active variables for each join is identified. Then the match lookup table from section 3.3 is altered so that its outputs are pruned of inactive variables. This is not difficult: given a left input with variables V and a right input with variables V', the output entry's variables are changed to active_vars$(V \cup V')$ instead of $V \cup V'$; the match lookup table is then populated accordingly. For example, suppose we are populating the lookup table at join J_{22} for left input $e_1 : \langle v = 1, w = 2 \rangle$ and right input $e_2 : \langle w = 2, x = 3 \rangle$: the table's dimensions are therefore $d^2 * d^2 = 9 * 9 = 81$. If we do not use the active variables optimization, then the output entry is $e_4 : \langle v = 1, w = 2, x = 3 \rangle$, and the values used to populate the table are $F(1,2) = 2$, $F(2,3) = 6$, and $F(1,2,3) = 6$, respectively. But if we do use the optimization, then the output entry is $e_4 : \langle x = 3 \rangle$, because only variable x is used in later computations. Therefore, the match lookup table would be populated with the values $F(1,2) = 2$, $F(2,3) = 6$, and $F(3) = 3$, i.e. we would set Match-Lookup$(2,3)$ to 3, instead of 6.

Of course, other network operations must be adjusted to reflect this change. For example, join J_{23} must be adjusted so that it expects a 2-ary left input instead of a 4-ary left input, and its indexing and matching tables allocated and populated accordingly. Also, in joins that prune away inactive variables from their outputs, two different pairs of inputs may produce the same output (e.g. entry $e_6 : \langle v = 2, w = 2 \rangle$ will also combine with entry e_2 to produce output entry e_4). The code must therefore be adjusted so that only the first copy of an output entry is passed onwards to the rest of the join network; subsequent copies are simply discarded.

When these adjustments are made, the active variables optimization offers dual benefits. First, by decreasing the size of entries throughout the network, it does the same for the various tables that operate upon them. So, active variables further mitigate any performance drain caused by overlarge tables. In addition, active variables eliminate an important source of isomorphic waste in the algorithm, as shown above.

3.6.1 Active Variables Results

In the test problems, the vast majority of duplicate entries are created after a series of joins, as fewer and fewer variables remain active and therefore more entries are effective duplicates. This causes fewer join targets to be made, but does not decrease the number of insertions greatly. Therefore, active variables are more effective in problems with longer clauses, such as LCL168-1 and BOO061-1, in which they have an effect early enough to prune significant matching in later joins. These problems also have many variables, so the ability of active variables to filter the number of variables needed in indexing and matching becomes important to reducing table sizes. These factors explain why active variables have an almost negligible effect in the easier problems, yet can increase speedup by over 4x in the harder problems.

However, though active variables mitigate the large table problem, one should not conclude that they make its effects negligible. My next optimization, Collection Match, is designed to make table size less relevant by compressing large numbers of entries into a smaller representation

3.7 Optimization 5: Inverse Representation

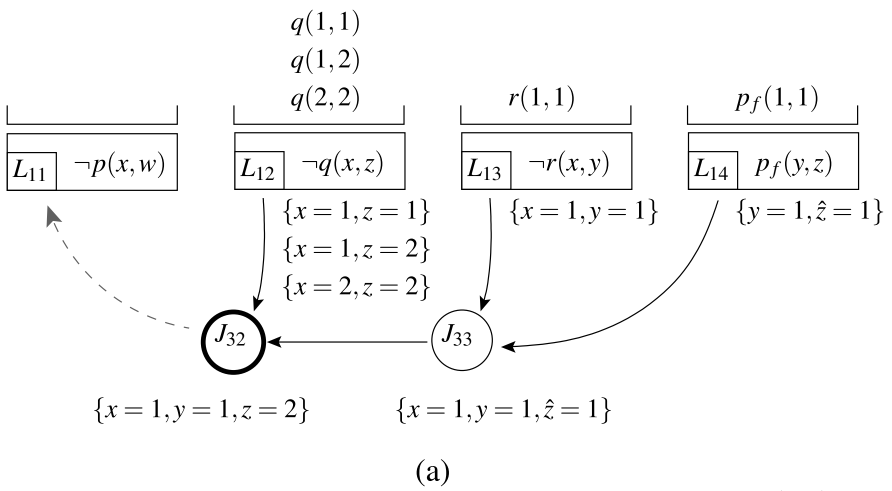
My most effective attempt at addressing the large table problem uses the insight that functions are one to one: that is, given $f(\overrightarrow{a})$, there is exactly one b such that $f(\overrightarrow{a}) =$

b. Heretofore, this requirement has been JOINSAT's bane, because it is required to explicitly assert $\neg p_f(\overrightarrow{x},c)$ for the $d-1$ domain elements c such that $b \neq c$. Each of these negated instances $\neg p_f(\overrightarrow{x},c)$ in turn match with non-negated literals p_f , creating the match blowup. However, the stringency of the one to one requirement allows an *inverse representation* to be used for these negated instances.

Given a non-negated literal p_f, the new method is to match the *non-negated* ground literal $p_f(\overrightarrow{a},b)$ with the clause literal, with the intent that in this context the ground literal actually represents the entire set of ground literals $\{p_f(\overrightarrow{a},c) : b \neq c\}$. The binding produced by this match is designed to have similar properties. For example, if our ground literal is $p_f(a_1,a_2,b)$ and our clause literal is $p_f(x,y,z)$, then the resulting bindset $\beta : \{x_1 = a_1, y = a_2, z = b\}$ actually stands for the set of bindsets $B : \{\{x_1 = a_1, y = a_2, z = c\} : b \neq c\}$. I say in this case that the binding $z = b$ is an *inverse binding*.

These semantics allow JOINSAT to make only one match at a non-negated literal where using normal matching it would have made $d-1$ matches. However, this semantics must be maintained through the process of joining with other literal nodes and also through the computation of least unvalued targets. I explain this process using an example, shown in Figure 3.10. Assume that $d = 4$ in the example and that the rightmost literal, $p_f(y,z)$, is an example of a functional predicate.

Both subfigures show level 3 of a join network. This level flows right to left and its rightward portion is the primary area in which large tables are joined. Figure 3.10 (a) shows matching in this level using inverse representation, while Figure 3.10 (b) shows matching using conventional representation. Figure 3.10 (a) portrays problem state immediately after the atom $p_f(1,1)$ has been assigned TRUE, meaning that the truth of $f(1) = 1$ has just been asserted. Before this optimization JOINSAT would eventually insert $\neg p_f(1,2)$, $\neg p_f(1,3)$ and $\neg p_f(1,4)$ into literal node L_{14} (as shown in Figure 3.10 (b)) as these negated atoms were assigned FALSE by the MACE2-hybrid component of JOINSAT. In the optimized system, these negated atoms are never matched against L_{14}; instead, the positive atom $p_f(1,1)$ is asserted. The resulting bindset is



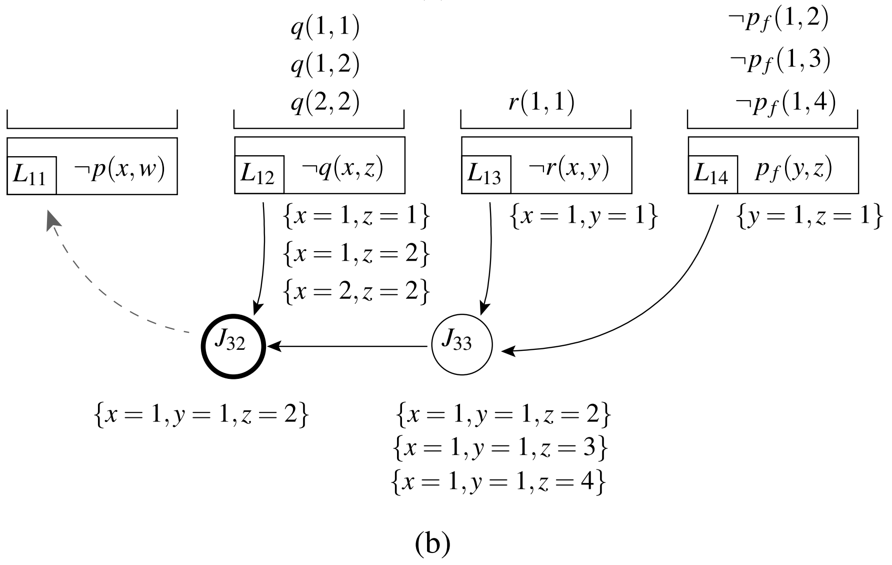


Figure 3.10: Match computation with (a) and without (b) inverse representation.

$\{y = 1, \hat{z} = 1\}$; I use $\hat{z}$ to show that the binding upon z is inverse. This bindset is passed on to join J_{23}, which matches upon variable y. Since $\hat{z}$ is not matched upon, it is simply copied to the match bindset like a conventional binding. However, next the bindset $\{x = 1, y = 1, \hat{z} = 1\}$ is passed to join J_{22}, which joins upon both z and x.

Joins match inverse variables as follows: if bindset β_1 contains a normal binding $z = a$, while bindset β_2 has an inverse binding $\hat{z} = b$, then the bindings iff $a \neq b$. The resulting match bindset will contain $z = a$, not $\hat{z} = b$. Therefore, when the bindset $\{x = 1, y = 1, \hat{z} = 1\}$is matched against the three bindsets from L_{12}: $\{x = 1, z = 1\}$, $\{x = 1, z = 2\}$, and $\{x = 2, z = 2\}$, the outcome is as follows. The first match fails because the bindings for z/$\hat{z}$ are both 1. The third match fails because the bindings for x are 1 and 2, respectively. The second match does succeed, because the bindings for $x$are each 1, while the bindings for z/$\hat{z}$ are 2 and 1, respectively.

Note that the match entries output by J_{22} are precisely the same as would be produced with the unoptimized system that inserts $\neg p_f(1,2)$, $\neg p_f(1,3)$ and $\neg p_f(1,4)$ into literal node L_{14}. In other words, using an inverse binding until it is naturally eliminated produces the correct semantics. However, it requires much less work: the number of intermediate entries, both in L_{14} and J_{23}, would have been more numerous using the conventional method, and this savings of course increases as domain and literal sizes increase. Generally speaking, using inverse bindings ensures that the number of ground literals matching with our non-negated functional literal will be proportional to 1, instead of $d - 1$: a great savings.

In the case in which inverse bindings reach the output of a selection or propagation join, special measures must be taken. Normally, we would populate a target vector for an entry containing binding $z = a$ by imposing that binding upon all target atoms. For example, if the target literal was $s(x, y, z)$, all atoms in the vector would have be of the form $s(x, y, a)$. For the case of an inverse variable, we do the opposite, producing an instantiation of $s(x, y, z)$ for every value of z except a.

It is also possible to have two non-negated functional literals, each containing a different inverse variable. It is certainly possible to implement a more complex semantics dealing with this possibility, but I only implement the simpler case described above. JOINSAT also refrains from using inverse literals in the case where the inverse variable is present in other arguments to the literal, e.g. $p_f(x,\hat{z},\hat{z})$: in this case a non-negated atom inserted by the system, e.g. $p_f(a,b,b)$, would stand for the set of all $\neg p_f(a,b,c)$ such that $b = c$ and $b \neq c$. In this case also, the system does not use inverse bindings and reverts instead to the standard matching method and bindings.

3.7.1 Inverse Representation Results

Inverse representations do appear to solve the large table problem, at least for functional literals, which are the primary source of the problem. Detailed analyses of the entries created at each node show that non-negated functional literal nodes using inverse representation create no more entries than their negated counterparts. For some problems this technique can achieve close to a 100% speedup. With this optimization, JOINSAT's speed finally approaches that of MACE2 on most problems.

However, the general problem of efficient join ordering remains important and unsolved. All versions of JOINSAT thus far set up join networks arbitrarily, without examining alternative orderings that might reduce matching. Therefore, inefficient joins are not avoided; for example, JOINSAT sometimes joins two literal nodes without any matching variables, resulting in a pure cross product of inputs. To reach the algorithm's full potential, some method of join ordering must be implemented.

3.8 Optimization 6: Static Analysis

My final optimization performs preprocessing on the flattened clauses, i.e. it is performed in the JOINSAT procedure (shown in Figure 2.9) immediately after the FLATTEN() procedure. This preprocessing step rearranges the clause literals to minimize the number of entries created by the join network.

I stated earlier (section 2.2.1) that avoiding the matching of "large tables" early in a sequence of joins tends to reduce the total number of entries created via matching.

So far, I have associated large tables with positive functional clause literals, and have explored several optimizations intended to reduce the impact of these literals. However, in the general case, any clause literal that is falsified by many ground literals is also "large". In practice, the more variables a clause literal contains, the more ground literals that falsify it. So, it is beneficial to avoid joining clause literals with many variables early on in a join sequence. Now, because the active variables optimization is used, inactive variables are also not used in further computations. Therefore, it is also beneficial to maximize the number of variables that become inactive early on in the join sequence.

Another important factor determining the total number of entries created is the number of variables joined at each join. Generally, each variable join reduces the total number of entries created at the join by a factor of the domain size d. For this reason, an additional goal is to construct join sequences in which as many variables as possible are joined early on.

A final determinant of the number of entries created is the use of inverse variables. Until an inverse variable is eliminated (converted into a standard variable), it reduces the number of entries by a factor of d (because $d-1$ entries are represented as one entry). When the inverse variable is eliminated, it effectively splits the one entry into $d$entries. Therefore, it is beneficial to defer the elimination of the inverse variable for as long as possible.

The heuristics above can sometimes conflict: orderings can boost performance for the one of the reasons above while slowing it for another. To properly compute the compound effect of the multiple factors identified above, the static analyzer uses a cost function modeling the number of entries created at every join network node for a given clause literal node ordering. This cost function is used as a heuristic by a randomized local search to explore a space of possible clause literal orderings: the best ordering is saved and used to order the clause literals.

3.8.1 Static Analysis Results

For some problems, the “original” clause orderings produced without static analysis are quite good, and in some cases seem optimal. For these problems, adding static analysis can actually decrease efficiency slightly. While the cost function and local search used produce very good results, they are not sophisticated enough to reliably find an optimal ordering (and I surmise that doing so is probably itself an NP-complete problem). The static analysis process also adds a small amount of overhead to the preprocessing phase.

This version of JOINSAT also imposes stricter limits on the number of derived clauses because large problems can cause very great numbers of derived clauses to be created. For some problems, this leads to missed propagation opportunities (this in particular accounts for the slow performance on problem cd). However, for most problems, the clause literal ordering output by static analysis leads to substantial improvement that more than makes up for the overhead and lost propagation opportunities.

3.9 Listing of Results for MACE2 and various JOINSAT versions

MACE2					
problems	time	#flips(K)	#updates(K)		
cd	0.03	84	742		
o1e1	19	15068	139368		
o1e4	27	21917	211635		
LCL168-1	21	54281	667597		
BOO061-1	232	308677	1055984		

JOINSAT - Base					
problems	time	#flips(K)	#entries(K)	#inserts(K)	
cd	0.15	1	94	93	
o1e1	270	1168	129291	107773	
o1e4	485	2658	237288	209869	
LCL168-1	6084	5735	1751942	>MAXINT	
BOO061-1	TIMEOUT				

JOINSAT - Collection Match					
problems	time	#flips(K)	#entries(K)	#inserts(K)	#interims(K)
cd	0.15	3	28	26	35
o1e1	53	2236	18777	14065	54137
o1e4	94	5309	31633	21803	97762
LCL168-1	220	12261	89716	94648	172070
BOO061-1	821.64	129614	579190	541862	912328

JOINSAT - Alternate Network					
problems	time	#flips(K)	#entries(K)	#inserts(K)	#interims(K)
cd	0.27	3	40	37	55
o1e1	56.74	2236	18373	12144	45500
o1e4	102	5309	30133	16888	80339
LCL168-1	285.82	12286	118696	123912	214702
BOO061-1	1449.65	130352	837692	836202	1493338

JOINSAT Static Analysis					
problems	time	#flips(K)	#entries(K)	#inserts(K)	
cd	0.14	54	260	312	
o1e1	19.35	3436	15514	18692	
o1e4	25.42	5838	18426	23332	
LCL168-1	33.45	12258	145954	257468	
BOO061-1	207.71	120578	715800	1045928	

Table 3.2: Experimental comparison of versions of JOINSAT (part 1). Some results are included in both parts of the table for easy reference. All times are in seconds.

MACE2					
problems	time	#flips(K)	#updates(K)		
cd	0.03	84	742		
o1e1	19	15068	139368		
o1e4	27	21917	211635		
LCL168-1	21	54281	667597		
BOO061-1	232	308677	1055984		

JOINSAT - Base					
problems	time	#flips(K)	#entries(K)	#inserts(K)	
cd	0.15	1	94	93	
o1e1	270	1168	129291	107773	
o1e4	485	2658	237288	209869	
LCL168-1	6084	5735	1751942	>MAXINT	
BOO061-1	TIMEOUT				

JOINSAT - Precomputation					
problems	time	#flips(K)	#entries(K)	#inserts(K)	
cd	0.04	3	40	58	
o1e1	34	2235	69128	62073	
o1e4	60	5308	135580	127136	
LCL168-1	61	8323	290827	382160	
BOO061-1	1742	35245	1793205	3004056	

JOINSAT-Active Variables					
problems	time	#flips(K)	#entries(K)	#inserts(K)	
cd	0.04	3	32	49	
o1e1	32	2235	36317	62045	
o1e4	56	5308	71444	127135	
LCL168-1	39	8374	149406	265345	
BOO061-1	892	34984	1190934	2028012	

JOINSAT Inverse Representation					
problems	time	#flips(K)	#entries(K)	#inserts(K)	
cd	0.03	3	31	49	
o1e1	22.84	2701	21526	33537	
o1e4	29.83	5839	27694	41866	
LCL168-1	60.13	12283	148160	262861	
BOO061-1	304.35	115646	782779	1180181	

JOINSAT Static Analysis					
problems	time	#flips(K)	#entries(K)	#inserts(K)	
cd	0.14	54	260	312	
o1e1	19.35	3436	15514	18692	
o1e4	25.42	5838	18426	23332	
LCL168-1	33.45	12258	145954	257468	
BOO061-1	207.71	120578	715800	1045928	

Table 3.3: Experimental comparison of versions of JOINSAT (part 2). Some results are included in both parts of the table for easy reference. All times are in seconds.

Chapter 4

Experimental Comparisons with Other Systems

This chapter compares JOINSAT performance to that of other state of the art satisfiability solvers. A version of JOINSAT incorporating the most powerful of the optimizations from the prior chapter is run on a test set, and then results are reported from other solvers for the same test set.

4.1 Experimental Problems and Methodology

I compare JOINSAT with five contemporary quantified satisfiability solvers: MACE2 [43], Paradox 1.3 [13], DarwinFM [5], iProver [23], and Geo [18]. The last four systems all did well in the recent CASC-21 theorem proving competition [61]; MACE2 was not entered in the competition, but has performed well in prior competitions. Two kinds of results are presented: first, a table of the total number of test problems successfully solved by each test system, and second, a series of graphs showing solution times for the various test systems compared with times for JOINSAT.

The test set is comprised of all eligible problems from the SEQ (Satisfiability with Equality) Division of the CASC-20 theorem proving competition [60]: 169 problems in all. The problems are available at the CASC-20 website; the problem names are given in appendix A.2. All of these problems are satisfiable. Using a large number of problems is a standard method of addressing one complication of evaluating competing systems. This is the fact, documented in [27], that the time required for a given satisfiability system to solve a given problem can strongly vary depending upon the initial search path selected. However, with a large test set, the superiority of one solver over another should be visible despite the random variation caused by the search path. To make these trends more visually clear, in each graph, the problems on the X axis are sorted, first by JOINSAT's solution time and then by the second solver's time, allowing easier visual comparison of the relative performance of the solvers. Note that JOINSAT and MACE2 tend to take the same search paths because each uses the same code for the larger DPLL search algorithm. As a result, signficant variance in their performance caused by the initial search path is less common.

The version of JOINSAT used for this experimental comparison is the one incorporating static analysis techniques from section 3.8. This version can sometimes be slightly slower than the inverse representation version from section 3.7. This is because the sorting benefit produced by the static analysis sometimes does not compensate for its additional overhead. However, the sorting benefit is important enough for more complex problems that the static analysis version of JOINSAT has the best overall performance.

These test runs were run on a different system from that used in the prior chapter because of the need to use the TPTPWorld testing software, which automates test runs for multiple problems and solvers. The system used for this section's results is a dual-core 1.60 GHz Pentium PC with 2Gb of RAM running Ubuntu Linux. Because of the large number of problems, I also use a much smaller timeout of 150 seconds of CPU time. Solvers can fail to solve a problem either by timing out or by giving up execution for some other reason (usually either exhaustion of system RAM or violation of hardcoded program limits on the number of clause variables). In results charts, these possibilities are presented graphically: timing out for a given problem is presented as a solution time of 150 seconds, while giving up execution is presented as a solution time of 200 seconds.

4.2 Results

I first present Table 4.1 showing the total number of test problems successfully solved by each test system. The table shows that JOINSAT does relatively well, solving slightly more problems than MACE2, and many more than iProver and Geo. When one con-

Solver	#Problems Solved (out of 169)
JOINSAT	103
MACE2	102
MACE2 w/o clause splitting	87
Paradox 1.3	160
DarwinFM	116
iProver	40
Geo	90

Table 4.1: Number of test problems solved by each tested system.

siders that both of the systems that solved more problems than JOINSAT (Paradox and DarwinFM) use the optimization of clause learning, while JOINSAT does not, JOINSAT's performance seems promising.

In the following sections, the solution times are presented in graph form: each graph compares JOINSAT's performance with one other system. In each graph, the results are sorted, first by JOINSAT's solution time and then by the second solver's time, allowing easier visual comparison of the relative performance of the solvers. A logscale Y axis is used to make significant performance differences more noticeable.

4.2.1 JOINSAT vs. MACE2

As mentioned in earlier sections, MACE2's [43] ground instantiation approach is the preeminent state of the art method for solving FQSAT problems. Because one of my goals in the dissertation is to show that my join network method is superior to the ground instantiation method, the experimental comparison of JOINSAT with MACE2 is the most important in this chapter. These results, shown in Figure 4.1, are hopeful but inconclusive. Those problems that are relatively easy for JOINSAT to solve (roughly, problems 1-80) are solved even more quickly by MACE2, often by more than an order of magnitude. This initial lag is chiefly explained by the low numbers of clause variables involved in these problems: only when there are many clause variables does MACE's instantiation method, which updates a number of ground clauses exponential in the number of clause variables, experience significant slowdown. Conversely, JOINSAT requires significant overheads from the static analysis and precomputation processes. Also, JOINSAT's matching process has much higher overhead per match than does MACE2's clause update process per update. So, I expect MACE2's performance to be superior on easier runs.

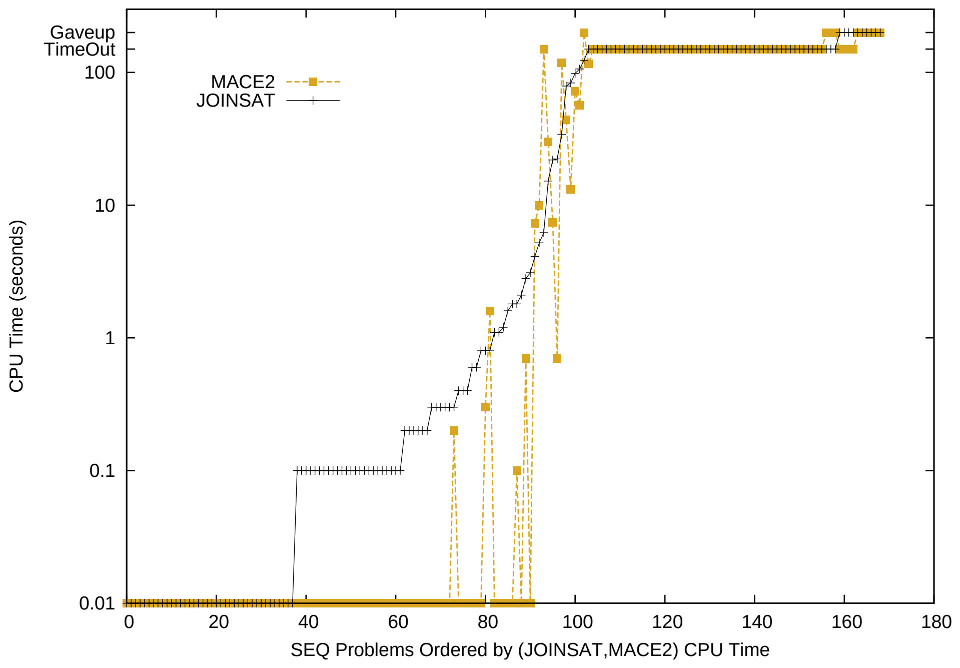


Figure 4.1: Comparison of solution times for JOINSAT and MACE2. Results are sorted by JOINSAT's solution time, and within that by MACE2's solution time. Timeouts and Gaveups are shown on the graph as 150sec and 200sec, respectively.

However, as the number of clause variables increase, JOINSAT's performance steadily overtakes MACE2's performance, until in problems 90-102 neither system seems to dominate. Unfortunately, neither system can solve more complex problems, making further comparison of the competing methods impossible. Recall that both MACE2 and JOINSAT use the DPLL search algorithm and that MACE2's ground instantiation method and JOINSAT's join network method only update the state at each iteration of the search. Therefore, total solution time complexity is still dominated by the number of search iterations, not by the time required to update at each iteration. For a number of those problems barely solvable by JOINSAT (i.e. problems 90-102), the number of search iterations is in the hundreds of millions, while the number of joins required at each iteration is one hundred or less, sometimes ten or less. If the overall search algorithm could be improved and more complex problems solved, there is reason to hope that JOINSAT would begin to dominate MACE2 with regard to run time. One project for future work along these lines would be to add optimizations to JOINSAT that reduce the number of search iterations (e.g. clause learning) and then test it against a similarly enhanced ground instantiation system.

At this point I can answer questions first raised in section 1.5.2 about the ground instantiation method's intrinsic complexity, as well as the question of whether the join network method represents an improvement. I should first mention that two design choices considerably alter the original methods, changing the answers to these questions. The first is the clause splitting optimization employed by MACE2 and described in section 1.7.2.7. Clause splitting drastically reduces the degree of clause blowup experienced by MACE2 as problems become more complex, making the solver require much less memory and run much faster. The second design choice is JOINSAT's use of precomputation. The memory required by precomputation lookup tables is generally even greater than that required by MACE2's tables of ground clause tallies, so JOINSAT is in this way as vulnerable to clause blowup as is MACE2. This memory usage may also reduce overall speed because of cache failures, but I cannot know this for sure without implementing a new version of JOINSAT that does not use precomputation (a system like the JOINSAT Base, but without the representational crudity of that early system). Nor can I determine at present whether a non-precomputed join network system would use less memory than a ground instantiation system enhanced with clause splitting (although I suspect this would be true).

Despite these qualifiers, my hypothesis that a join network would require fewer matches than an instantiation method would require clause updates seems to be true, even for problems in which join networks are slower than ground instantiation, as shown in Table 3.3.

A related question of interest is: how powerful is the clause splitting optimization employed by MACE2 and described in section 1.7.2.7? In Figure 4.2, the performance of JOINSAT is compared with that of a version of MACE2 in which clause splitting is disabled. The results show that the overhead of clause splitting slows MACE2 performance for easier problems (problems up to about problem 80), but thereafter is a source of very significant speedup, sometimes more than two orders of magnitude. From these results, it is clear that MACE2 would be substantially inferior to JOINSAT for harder problems were it not for clause splitting, a relatively recent innovation. So, one may hope that over time similar optimizations may be found for the basic join network method.

4.2.2 JOINSAT vs. Paradox 1.3

The next comparison is between JOINSAT and Paradox 1.3 [13] (described in section 5.1), which won the SAT Division at the 2007 CASC-J3 [61] and is probably the fastest quantified satisfiability solver available at the time of this writing. Paradox significantly outperforms JOINSAT on almost all problems, often by several orders of magnitude. This is not unexpected, and it is important to place these results in context. First, Paradox uses key optimizations like clause learning and conflict-directed backtracking that are not included in JOINSAT (but which could be added). Second, Paradox is built upon a SAT solver using a MACE2-style ground instantiation representation, so Paradox's overall performance could still be boosted by replacing the SAT solver with JOINSAT. So, to the degree that JOINSAT improves upon MACE2's performance, JOINSAT could enhance Paradox.

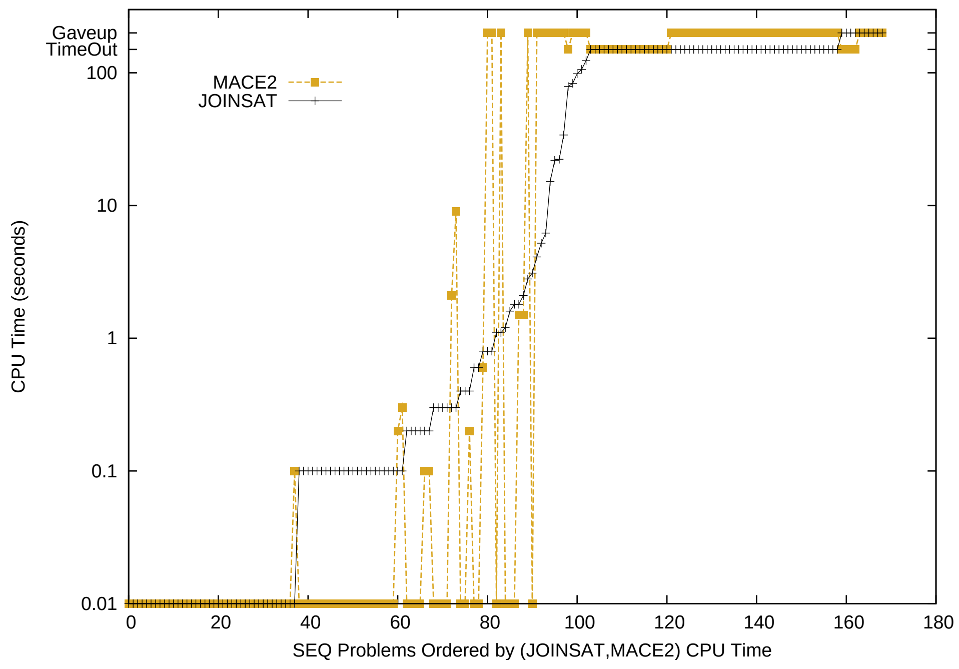


Figure 4.2: Comparison of solution times for JOINSAT and MACE2 with MACE2's clause splitting disabled. Results are sorted by JOINSAT's solution time, and within that by MACE2's solution time. Timeouts and Gaveups are shown on the graph as 150sec and 200sec, respectively.

4.2.3 JOINSAT vs. DarwinFM

Unlike Paradox, the final three solvers (DarwinFM, iProver, and Geo) do not use a MACE2-style SAT solver as a back end. DarwinFM [5] does use some of the same preprocessing steps as MACE2 and JOINSAT (e.g. eliminating function instances via flattening and adding constraints to enforce functional properties, as described in section 1.7.2). However, DarwinFM uses the Darwin solver (described in section 5.2.1) to determine the satisfiability of the quantified clauses output by preprocessing.

The comparative results (showin in Figure 4.4) show that DarwinFM solves significantly more problems than does JOINSAT. This is actually not entirely accurate, as JOINSAT gives up on many problems because it adheres to a limit on the total number

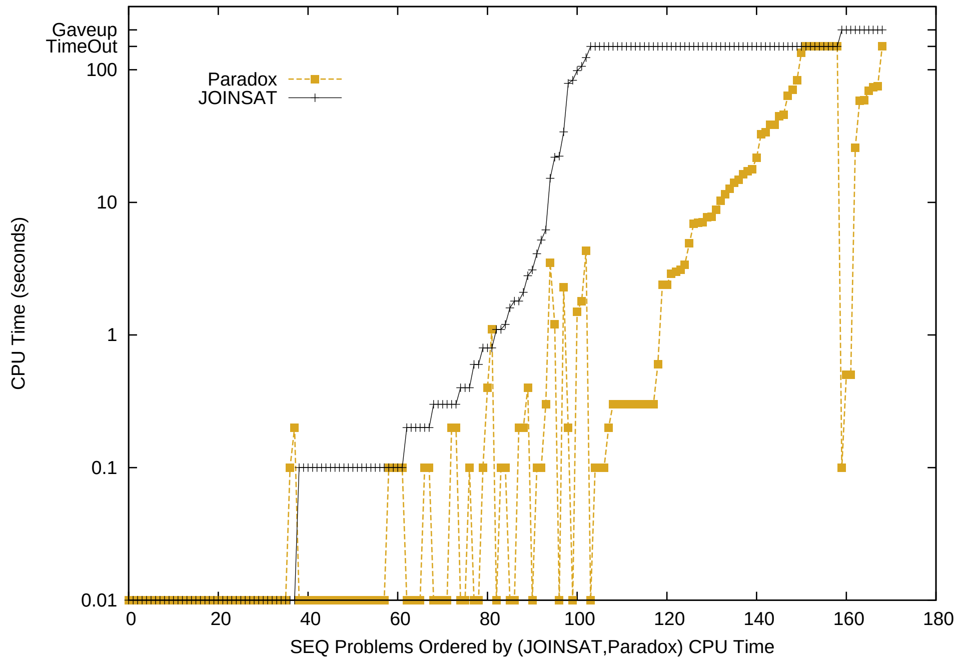


Figure 4.3: Comparison of solution times for JOINSAT and Paradox 1.3. Results are sorted by JOINSAT's solution time, and within that by Paradox's solution time. Timeouts and Gaveups are shown on the graph as 150sec and 200sec, respectively.

of clause variables imposed by MACE2. I kept this limit to avoid distortion of the results chart comparing MACE2 and JOINSAT, but it is not intrinsic to the join network approach.

Considering only those problems solved by both systems, neither shows consistently better runtime performance. This is somewhat surprising, given that Darwin (the back end system) uses clause learning, which usually provides speedup of several orders of magnitude. Also, as noted in section 5.2.1, Darwin's ability to add quantified literals should be an advantage. It is my belief (although I have not tested this) that the answer to this riddle lies in the comparative overhead between JOINSAT's and Darwin's approaches to subsearch. JOINSAT uses join networks to identify prospective atoms to flip. Because these networks simply match against ground atoms and then pass the resulting bindings through the network, they can be implemented reasonably simply.

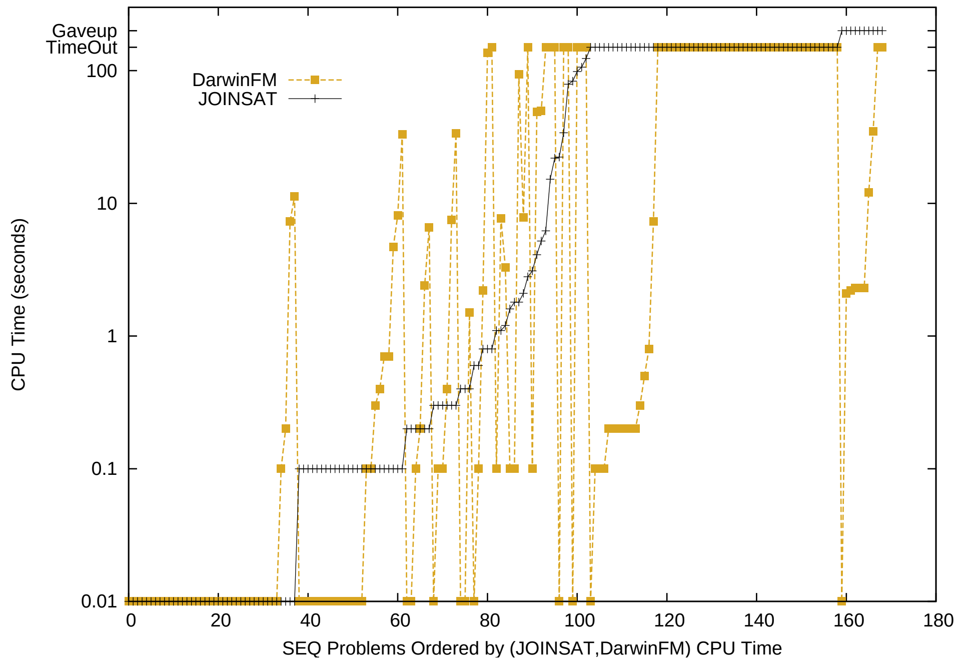


Figure 4.4: Comparison of solution times for JOINSAT and DarwinFM. Results are sorted by JOINSAT's solution time, and within that by DarwinFM's solution time. Timeouts and Gaveups are shown on the graph as 150sec and 200sec, respectively.

In particular, the implementation is simple enough that it can be precomputed, achieving significant speedup. In contrast, although Darwin performs in principle a similar matching between literals and clauses, the literals may be non-ground. As a result, the matching process is much more robust and complex, unifying arbitrary quantified literals and storing the result in a tree caching instantiation and generalization relationships. I believe that the additional overhead of this matching process prevents DarwinFM from consistently outperforming JOINSAT.

4.2.4 JOINSAT vs. iProver

iProver [23] is an instantiation-based theorem prover than can also be used to compute satisfiability for quantified clauses. iProver is distinguished from the systems mentioned earlier in the chapter because it operates by resolving theory clauses together to produce new clauses; the solvers mentioned earlier instead resolve literals from the current assignment or context with theory clauses. However, iProver is also different from many resolution-based theorem provers because the resolved clauses it produces are always subsumed by the resolvents; in other words, the resolution process produces ever more specific clauses. Once a resolved clause is ground, iProver uses a ground SAT solver to check its satisfiability.

It is difficult to explain why JOINSAT (as well as the other tested solvers) outperform iProver. However, probably the answer lies less with JOINSAT's innovative subsearch approach and more with the overall efficiency of the DPLL algorithm. Satisfiability provers using resolution between clauses, as iProver does, have not fared particularly well in recent years when matched against DPLL-style provers.

4.2.5 JOINSAT vs. Geo

Geo [18] is a solver that reasons with *geometric formulas*, an equisatisfiable fragment of first order logic first introduced by Skolem. This fragment is similar in function to the quantified clauses used by JOINSAT and DarwinFM: functions are translated away, and

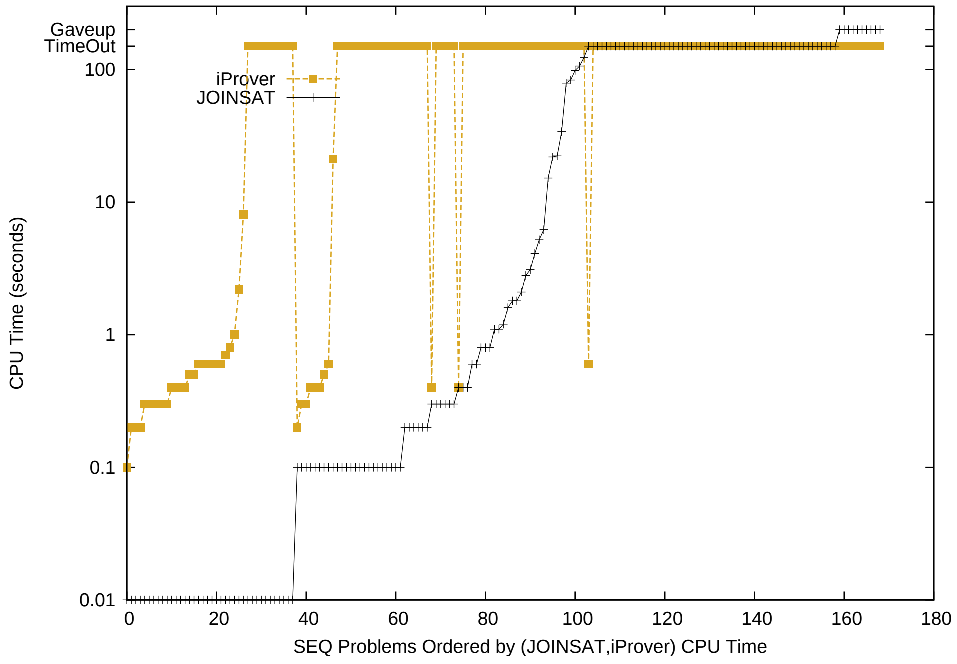


Figure 4.5: Comparison of solution times for JOINSAT and iProver. Results are sorted by JOINSAT's solution time, and within that by iProver's solution time. Timeouts and Gaveups are shown on the graph as 150sec and 200sec, respectively.

existential quantifiers are isolated from the rest of the logical content (JOINSAT and Darwin instead translate existential quantifiers into function instances). Like iProver, Geo resolves theory formulas together to produce new formulas. Geo also incorporates a form of learning conflict-driven geometric formulas. JOINSAT performs quite well against Geo, which again is surprising considering that JOINSAT does not use clause learning. The authors of Geo expect to produce a much more optimized system in future versions.

4.3 Summary

In this section, JOINSAT was tested against five contemporary solvers. JOINSAT clearly outperformed two of these solvers, iProver and Geo. JOINSAT was outpaced by two

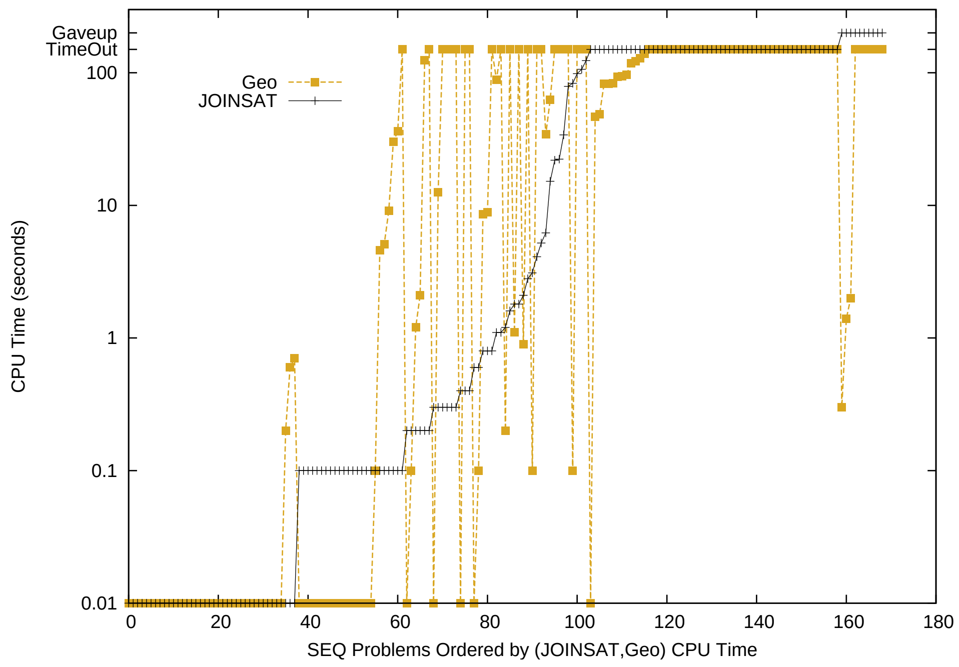


Figure 4.6: Comparison of solution times for JOINSAT and Geo. Results are sorted by JOINSAT's solution time, and within that by Geo's solution time. Timeouts and Gaveups are shown on the graph as 150sec and 200sec, respectively.

solvers, Paradox and DarwinFM, but both of these solvers use optimizations that could be added to JOINSAT in the future, particularly clause learning, which provides a significant search-space pruning advantage. JOINSAT showed performance roughly comparable to the final solver, MACE2: JOINSAT solved one more problem than did MACE2, but MACE2 solved less challenging problems more quickly than did JOINSAT. The comparison with MACE2 is the most important for evaluating my method, because MACE2 exemplifies the ground instantiation method I hope to improve upon. At present, JOINSAT cannot be said to be markedly superior to MACE2, although I hope to achieve this through future optimizations.

Chapter 5

Related Work

In this section, alternative approaches to solving quantified satisfiability problems are mentioned. This presentation is separated into two sections. The first presents other FQSAT solvers. The second introduces a different problem formalism, QSAT, and then examines QSAT solvers.

5.1 FQSAT Research

The early years of FQSAT research revolved around efforts to convert the FQSAT problem to SAT by translating away all quantifiers and functions, and then solve using a SAT solver. The MACE2 solver [43] was one of the earliest solvers using this method; it is examined in detail in section 1.7. Other solvers explored optimizations for the ground instantiation method. For example, the SEM solver [65] used a ground solver integrated with special data structures to handle function instances in the theory: it had some success attacking mathematical problems dominated by equality and functions, such as quasigroup existence problems.

In recent years, FQSAT solvers using entirely new technologies, and in some cases new representations, have proliferated. The Lifted WSAT solver [26] is an early attempt to compute FQSAT using quantified clauses, without a ground conversion. Like all SAT and QSAT solvers, Parkes' and Ginsberg's method still constructs assignments in a ground fashion, adding or deleting one ground atom at a time. However, the problem of computing the degree to which the assignment satisfies the theory (labeled *subsearch* in [26]and discussed in more detail in section 1.5) is performed quite differently. For this purpose, the quantified clauses are converted into a Constraint Satisfaction Problem

(CSP), with a twist: a solution to the CSP represents some ground clause instantiated from the theory that is made false by the assignment. This and similar CSP solvers are used to compute problem state at each iteration. A disadvantage of this method is that the CSP solver's variables range over all values of the domain, so computing solutions to it can take exponential time in domain size. This contrasts with my approach, which has complexity directly proportional to the number of atoms in the assignment set, and is potentially independent of the size of the domain.

The ZAP solver [19] uses special properties of mathematical groups to eliminate isomorphic structure in common FQSAT problems. Its key insight is that collections of sets of permutations upon numbers are groups, and therefore representable by a set of group generators that is logarithmic in the size of the group. Now, as we saw in section 1.3.2, a FQSAT clause can be instantiated to a set of grounded propositional clauses (e.g. equation 1.3). If we consider an arbitrary clause from this set, we may think of all the other members as permutations of the first, in that different domain values are substituted for the ground terms in the first clause. Therefore, an FQSAT clause can be represented as a pairing (c, G) of this first ground clause c with a group G representing the permutations. Since groups can be represented compactly using generators, so can FQSAT clauses, as well as literals used to resolve with them. ZAP's inference procedure is therefore a version of DPLL using a special resolution computation upon these augmented clauses, a computation that in theory could perform many conventional resolution steps at once. An added advantage is that the space required to maintain problem state should be very low. The algorithm appears to achieve these theoretical gains for some historically combinatorial SAT problems (e.g. pigeonhole problems), but has not been entered in satisfiability competitions or been made publicly available, making a serious evaluation difficult. If ZAP's theory can be made a reality, it might be a major advance over my algorithm, which is inspired by the potential efficiency of database-style joining, and does not use advanced compression formalisms.

A final relevant FQSAT system is Paradox [13], which is technically a QSAT solver, although it is limited to effectively propositional theories. This system solves QSAT problems by first determining an upper bound d_{MAX} (possibly infinite) for the size of the domain universe, beyond which any search efforts are redundant. It then tries to solve the problem MACE2-style, by conversion to SAT, beginning with domain size $d = 1$ and iterating all the way to d_{MAX} until a solution is found (or not). Paradox demonstrates that at least for certain classes of QSAT, there is an overlap of QSAT, FQSAT and FQSAT methods. Paradox' performance seems superior to MACE2's, partially because Paradox improves upon clause splitting and symmetry reduction techniques used by MACE2. Paradox also reuses search information from prior domain sizes and is able to infer variable sorts. An important contributor to Paradox's efficiency is its ground SAT solver, MiniSAT, which uses various optimizations including clause learning.

5.2 The Quantified Satisfiability (QSAT) Problem

QSAT is the problem of satisfiability for finite theories of first order logic (FOL) sentences. It is relevant to my work because most satisfiability problems of interest can be formalized in any of SAT, FQSAT, or QSAT: the question of which formalism can be solved most efficiently naturally arises.

To present QSAT, I first review definitions needed to express a model in a succinct fashion. A *Herbrand universe* $\mathscr{H}$ for a language $\mathscr{L}$ is the set of all ground terms of $\mathscr{L}$ (terms constructable from the predicates, function and constant symbols of $\mathscr{L}$). The *Herbrand base* is the set of all ground atoms constructable from L using only terms from the Herbrand universe. A *Herbrand assignmen*t for $\mathscr{L}$ is a logical assignment (a pairing of a universe and an interpretation function) in which the universe is $\mathscr{H}(\mathscr{L})$ and the interpretation maps every constant or function instance to itself, i.e. $\mathscr{I}(f(t_1,\ldots,t_n)) = f(t_1,\ldots,t_n)$. Herbrand assignments are significant because Herbrand's theorem proves any FOL sentence is satisfiable if it has a Herbrand model. Typically, a Herbrand model is expressed as an assignment mapping every member of the Herbrand base to one of $\{$TRUE,FALSE$\}$, and sometimes is even more concisely represented as just that subset of the Herbrand base that is assigned TRUE.

Problem. Quantified Satisfiability (QSAT).

Given:

1. a finite theory $\mathscr{T}$ of FOL sentences;
2. implicitly, a language $\mathscr{L}$ containing the predicates, constants, and function symbols in $\mathscr{T}$;

return either:

1. a Herbrand model for $\mathscr{T}$; or
2. FAILURE, if no such model exists.

As a decision problem, QSAT is equivalent to the problem of validity for first-order theories, i.e. deciding whether theory $\mathscr{T}$ logically implies sentence θ (written $\mathscr{T} \models \theta$) . To see this, consider that $\mathscr{T} \models \theta$ iff $\mathscr{T} \cup \{\neg\theta\}$ is not satisfiable. Now, validity for FOL theories is semi-decidable in the general case (Church-Turing Theorem, 1936), that is to say, there exist algorithms such that $\mathscr{T} \models \theta$ iff the algorithm returns yes in finite time, but there exists no algorithm such that $\mathscr{T} \not\models \theta$ iff the algorithm returns no in finite time. So, typically an ATP/QSAT solver will guaranteeably return an answer if $\mathscr{T} \models \theta$, but may not not halt if $\mathscr{T} \not\models \theta$. The problems remain distinct in the sense that typically solvers are constructed to return a useful answer to only one of the problems: an ATP solver returns a proof when $\mathscr{T} \models \theta$, and returns FAILURE (sometimes) otherwise, while a QSAT solver returns a Herbrand model (sometimes) when $\mathscr{T} \not\models \theta$, and returns failure otherwise..

5.2.1 QSAT Research

The problem of FOL validity is better known as automated theorem proving (ATP), although this term does cover other problems and other logics. The main (though by no means only) algorithmic methods in ATP concern resolution (the ur-algorithm for which is Robinson's resolution solver [51]; recent competitive systems include Otter [44] and Vampire [50]) and tableaux methods (recent competitive systems include SETHEO [46] and DCTP [40]). Of more relevance to us are recent methods in *instantiation-based* theorem proving. The most interesting example of these is the Model Evolution Calculus (MEC) [8,6] .

Darwin, the implementation of the Model Evolution Calculus, is a solver whose algorithmic structure parallells that of DPLL, but yet has the capability to reason with quantified clauses and literals. Darwin evolves a compact representation of a Herbrand assignment called a *context*. The context is a set of (possibly non-ground) literals; each literal is held to be true except insofar as some other more specific form of the literal conflicts with it. For example, a context might contain $\{p(x), \neg p(c)\}$, which is interpreted as that every specialization of $p(x)$in the Herbrand base (e.g. $p(a), p(b), p(f(a))$, $p(f(f(a))), \ldots$) is TRUE, except for $p(c)$, which is FALSE. The context is progressively added to until it satisfies every quantified clause.

The Model Evolution Calculus is significant to this dissertation chiefly as an alternate approach to computing satisfiability using a quantified representation. While my method uses a quantified representation of the clauses, it represents an assignment as a set of ground literals, and grows the assignment only one ground literal at a time. This may be less efficient than Darwin, which can add quantified literals to its context: for example, Darwin could add $p(x,y,z)$ to its context, while my solver is forced to add every ground instantiation of $p(x,y,z)$. However, presumably Darwin requires a substantial overhead to represent sets of arbitrarily complex literals and compute their unification. Darwin won top honors in the most recent Conference for Automated Deduction ATP System Competition ([61]) in the EPR division: finding models for effectively propositional quantified formulas (formulas whose Horn universes are finite). Darwin's performance is undoubtedly enhanced by its use of the clause learning optimization (described in [7]).

Chapter 6

Conclusion and Future Work

6.1 Future Work

JOINSAT's performance has already improved by a factor of 300x since its first official version (the base JOINSAT algorithm), and this algorithm in turn achieved a 20x speed gain over the very early experimental version of the algorithm. A number of prospective optimizations might continue that trend.

6.1.1 Shared Indices

One important optimization would involve sharing structure between the many join indices in the network whose inputs are from literal nodes. Since most of these are upon very small numbers of variables, and since two indices often share the same variables, they could be represented as permutations of one another and represented only once in the system. This is especially true for those problems having ten or twenty literal nodes with the same predicate: surely insertions of these nodes entries involves significant duplication. To some degree, MACE2 can avoid this duplication because in preprocessing it analyzes the set of ground clauses and prunes those which subsume each other after any unit clauses are resolved.

Insertions from literal nodes into indices are a very significant component of the overall work, so reducing isomorphism in this area could bring great efficiency gains.

6.1.2 Avoiding duplicate entries

Section 3.6 discussed the pruning of redundant match entries using active variables. Though this technique creates considerable speedup, the computation of the match entries before they are thrown away requires very significant time. For example, in most of the experimental problem instances, the number of duplicate entries created is as large as the number of non-duplicate entries. Furthermore, the creation of the vast bulk of these duplicates typically occurs in a very few join nodes at the end of a series of joins. At these joins, it is not uncommon for the ratio of duplicate to non-duplicate entries to approach $d/1$. Great savings could be achieved by avoiding the computation of these duplicates, instead of computing them and then throwing them away.

Accordingly, I have formulated (but not yet implemented) a join strategy that never computes these duplicates in the first place. This involves a completely different match algorithm that departs from the usual practice of scanning right inputs to find those that match with a new left input. Instead, the set of all possible *output* match entries is tracked to see which of these have never before been output. When this set grows sufficiently small (as it does at the joins in question), the system can quickly check for each set member if the current input can produce it. This method may lead to considerable savings in match time; the risk, of course, is that the additional overhead may overwhelm the savings.

6.1.3 Reducing table sizes

An important performance issue for JOINSAT is the large size of the tables computing indexes and match entries. These tables can grow very large, especially because JOINSAT eschews the clause splitting technique used by MACE2. It is something of an irony that JOINSAT, which began as an effort to avoid ground instantiation, presently uses much greater amounts of RAM than does MACE2. I have already detailed several efforts to use parsimonious representations that (among other benefits) reduce table size. Further research in this area could improve memory usage and thereby reduce cache misses, imprving performance.

6.1.4 Static and adaptive join ordering

At the conclusion of the discussion of results, I showed that a static analysis of join order could boost performance slightly past the current state of the art. The kind of syntactic static analysis performed by JOINSAT is crude; far more sophisticated techniques are well understood [54]. More sophisticated adaptive analyzers [4, 29] could increase efficiency even further by reordering the network based upon statistics from an ongoing run.

6.1.5 Backing away from precomputation

A different dimension of improvement for JOINSAT lies in finding ways to make a non-precomputed version perform (nearly) as well as a precomputed version.

The original intended use of the join network method was to represent and solve AI planning problems using quantified satisfiability. While various classical planning problems have been solved using SAT methods [37,38,64], it seems less likely that SAT can be used to tackle temporal and resource planning. The problem is that first, that relatively complex sentences, with many variables, are required to model planning, and second, that often the domains of these problems (e.g. the number of planes or trucks or steps) are large. As demonstrated in section 1.7.3, a ground SAT representation is exponential in the number of clause variables, and can also bog down even for small clauses if the domain size is large. These problems are sidestepped by the Blackbox SAT planner [38] by its use of a simplifying (planning graph) formalism for the planning problem, but it is unclear whether planning graphs can be used for more complex temporal and resource problems [58].

If any satisfiability solver is to solve these complex planning problems, that solver's complexity must be proportional to the assignment size, and thereby at least partially independent of the domain size. In this sense, precomputation may be seen as a step backward for a solver like JOINSAT, because precomputation requires space d^v, where d is the domain size and v is the number of active clause variables. Techniques that might help achieve an efficient non-precomputed solver include multidimensional indexing and more compact entry representations, especially those that effectively use one copy of an entry to represent its often myriad instances within the network.

6.1.6 Applying State of the Art SAT Optimizations

Finally, if JOINSAT can be made substantially more efficient than ground methods, a wide field of application comes into view. Even a precomputed version of JOINSAT could be used in place of SAT in many of its research and industrial uses. A non-precomputed version would be particularly attractive for domains like AI Planning because it would combine the recent explosive gains in satisfiability performance with a tolerance for very large domains.

Exploring these applications would require the lifting of state of the art SAT optimization techniques like conflict learning, non-chronological backtracking, and atom choice heuristics to a quantified representation. Since these techniques seem to require no more information than that computed by join networks, their adaptation looks promising.

6.2 Contribution and Conclusions

This dissertation introduced a new method of computing quantified satisfiability problems. After extensive optimization, this method was shown to be competitive with the state of the art method of ground clause instantiation and update. It was also demonstrated that in many cases the method is less complex in a theoretical sense. For some problems, JOINSAT requires an order of magnitude fewer inserts (its dominant cost task) than MACE2 requires clause updates (the dominant cost task for ground solvers).

The dissertation makes several technical contributions. The most important of these is the adaptation of the Rete method to compute subsearch. This required the development of an underlying bindset semantics as well as mechanisms and networks for literal choice and unit propagation. These formalisms serve as a foundation upon which further research in join-based satisfiability can be conducted.

The technical contributions also include several powerful optimizations of the basic join network method that increased empirical efficiency. The precomputation method is significant not only for its resulting speedup, but also as a way of translating production system methods, traditionally very intensive consumers of dynamic memory, into algorithms using more static representations. The adaptation of Collection Match, while not empirically successful, uses a more compact representation than does the original CM algorithm, and may yet be improved to empirically live up to its theoretical superiority over traditional matching. The use of inverse representations solved a difficult problem for join networks, that of dealing with non-negated functional predicates efficiently. It may be possible to extend this technique to other instances in which many sets of bindings correspond to one inverse binding.

At present the join network method does not enjoy clear superiority over ground instantiation methods. However, we are to remember that the crucial optimization technique for MACE2 of clause splitting only appeared in 2003 [13]; before that time, ground FQSAT solver performance was orders of magnitude slower. Similarly, optimizations for JOINSAT may yet be discovered that make it the preferred method for solving satisfiability problems.

Appendix A

Test Problems

A.1 Problems Used to Compare JOINSAT Versions

A.1.1 cd

```
% benchmark parameters -n2 -N6 -p

% The formulas below are theorems of the equivalential calculus.
% We can show that neither is a single axiom, because e(x,x) does not
% follow.

list(usable).

-P(e(x,y)) | -P(x) | P(y).

% P(e(e(x,y),e(e(y,z),e(x,z)))).     % easier
P(e(e(x,y),e(e(y,z),e(z,x)))).     % harder

-P(e(a,a)).

% The following can speed up finding models for CD problems.
% It says that the all values for which P is false come before
% any of the values for which P is true.
% But it can be incomplete if you use -c or if you DON'T use
% -z 0.  In this case it works, because a gets assigned 0,
% and there is a model in which P(a) is false.

% -P(x) | -(x < y) | P(y).

end_of_list.

%list(mace_constraints).
%assign(a,0).
%end_of_list.
```

```
***********************

-e(v0,v1,v2) OR -P(v2) OR -P(v0) OR P(v1)
-e(v0,v1,v2) OR -e(v3,v0,v4) OR -e(v4,v2,v5) OR -e(v1,v3,v6) OR -e(v6,v5,v7) OR P(v7)
-e(v0,v0,v1) OR -P(v1) OR -a(v0)
-e(v0,v1,v0) OR -P(v0) OR P(v1)
```

A.1.2 o1e1

```
% benchmark parameters -N10 -p
%
% There is a model of size 10; it takes 275 seconds on a PIII933.
%
% We have an equation that holds for orthomodular lattices,
% and we show that it does not hold for ortholattices by
% finding an ortholattice in which the equation does not hold.
%
% The problem is from Norm Megill, September 1997.
%
% The equation is (E1) from
%
% @article{ ortholattice,
%    author = "W. McCune",
%    title = "Automatic Proofs and Counterexamples for some
%                  Ortholattice Identities",
%    journal = "Information Processing Letters",
%    year = 1998,
%    volume = 65,
%    pages = "285--291"}
%
% See http://www-unix.mcs.anl.gov/~mccune/papers/ortholattice/

include("ortholattice").

list(usable).

% The original denial:
%
% c((A ^ c(B)) v c(A)) v ((A ^ c(B)) v ((c(A) ^ ((A v c(B)) ^
% (A v B))) v (c(A) ^ c((A v c(B)) ^ (A v B))))) != A v c(A).

% An equivalent denial that names subterms:  (This is necessary, because
% MACE cannot handle big equations.)

A ^ c(B) = D1.
A v c(B) = D2.
A v B = D3.
c(A) = D4.
D2 ^ D3 = D5.
D4 ^ c(D5) = D6.
D4 ^ D5 = D7.
```

```
D7 v D6 = D8.

c(D1 v D4) v (D1 v D8) != 1.

end_of_list.


****************************

Symbol  0, relation, Dummy_symbol/0, atoms    1 --   1,          0.
Symbol  1, relation,      =/2, atoms    2 --   5,          1.
Symbol  2, function,      ^/3, atoms    6 --  13,          0.
Symbol  3, function,      v/3, atoms   14 --  21,          0.
Symbol  4, function,      c/2, atoms   22 --  25,          0.
Symbol  5, function,     D1/1, atoms   26 --  27,          0.
Symbol  6, function,      B/1, atoms   28 --  29,          0.
Symbol  7, function,      A/1, atoms   30 --  31,          0.
Symbol  8, function,     D2/1, atoms   32 --  33,          0.
Symbol  9, function,     D3/1, atoms   34 --  35,          0.
Symbol 10, function,     D4/1, atoms   36 --  37,          0.
Symbol 11, function,     D5/1, atoms   38 --  39,          0.
Symbol 12, function,     D6/1, atoms   40 --  41,          0.
Symbol 13, function,     D7/1, atoms   42 --  43,          0.
Symbol 14, function,     D8/1, atoms   44 --  45,          0.

1. -^(v0,v1,v2) ^(v1,v0,v2)
2. -v(v0,v1,v2) v(v1,v0,v2)
3. -v(v0,v1,v2) -v(v3,v2,v4) -v(v3,v0,v5) v(v5,v1,v4)
4. -c(v0,v1) c(v1,v0)
5. -^(v0,v1,v2) v(v0,v2,v0)
6. -c(v0,v1) -c(v2,v3) -v(v3,v1,v4) -c(v4,v5) ^(v2,v0,v5)
7. ^(v0,v0,v0)
8. v(v0,v0,v0)
9. -c(v0,v1) v(v1,v0,1)
10. -c(v0,v1) ^(v1,v0,0)
11. v(1,v0,1)
12. v(v0,1,1)
13. ^(1,v0,v0)
14. ^(v0,1,v0)
15. ^(0,v0,0)
16. ^(v0,0,0)
17. v(0,v0,v0)
18. v(v0,0,v0)
19. -D1(v0) -B(v1) -c(v1,v2) -A(v3) ^(v3,v2,v0)
```

```
20. -D2(v0) -B(v1) -c(v1,v2) -A(v3) v(v3,v2,v0)
21. -D3(v0) -B(v1) -A(v2) v(v2,v1,v0)
22. -D4(v0) -A(v1) c(v1,v0)
23. -D5(v0) -D3(v1) -D2(v2) ^(v2,v1,v0)
24. -D6(v0) -D5(v1) -c(v1,v2) -D4(v3) ^(v3,v2,v0)
25. -D7(v0) -D5(v1) -D4(v2) ^(v2,v1,v0)
26. -D8(v0) -D6(v1) -D7(v2) v(v2,v1,v0)
27. -D8(v0) -v(v1,v0,v2) -D4(v3) -D1(v1) -v(v1,v3,v4) -c(v4,v5) -v(v5,v2,1)
```

A.1.3 o1e4

```
% benchmark parameters -N10 -p

include("ortholattice").

list(usable).

% This is the denial in which ground terms are "named".

c(A) = d2.
B v d2 = d3.
B ^ d2 = d4.
c(d3) = d5.
d2 ^ c(B) = d8.
d3 ^ A = d9.
d8 v d4 = d10.
d10 v d9 = d11.
d11 ^ d2 = d12.
d11 v d2 = d13.
c(d11) ^ d2 = d14.
d13 ^ A = d16.
d14 v d12 = d17.
d17 v d16 = d18.
d5 ^ d18 = d19.
c(d18) ^ d5 = d20.
d18 v d5 = d21.
d21 ^ d3 = d22.
d19 v d22 = d23.
d23 v d20 != 1.

end_of_list.

% list(usable).  % Original denial.
%     ( ( c( c(A) v B ) ^ ( ( ( c(A) ^ ( ( ( c(A) ^ B
%     ) v ( c(A) ^ c(B) ) ) v ( A ^ ( c(A) v B ) ) ) ) v ( c(A)
%     ^ c( ( ( c(A) ^ B ) v ( c(A) ^ c(B) ) ) v ( A ^ ( c(A) v
%     B ) ) ) ) ) v ( A ^ ( c(A) v ( ( ( c(A) ^ B ) v ( c(A) ^
%     c(B) ) ) v ( A ^ ( c(A) v B ) ) ) ) ) ) ) v ( c( c(A) v B
%     ) ^ c( ( ( c(A) ^ ( ( ( c(A) ^ B ) v ( c(A) ^ c(B) ) ) v
%     ( A ^ ( c(A) v B ) ) ) ) v ( c(A) ^ c( ( ( c(A) ^ B ) v
%     ( c(A) ^ c(B) ) ) v ( A ^ ( c(A) v B ) ) ) ) ) v ( A ^ (
%     c(A) v ( ( ( c(A) ^ B ) v ( c(A) ^ c(B) ) ) v ( A ^ ( c(A)
%     v B ) ) ) ) ) ) ) ) ) v ( ( c(A) v B ) ^ ( c( c(A) v B )
%     v ( ( ( c(A) ^ ( ( ( c(A) ^ B ) v ( c(A) ^ c(B) ) ) v ( A
%     ^ ( c(A) v B ) ) ) ) ) v ( c(A) ^ c( ( ( c(A) ^ B ) v ( c(A)
%     ^ c(B) ) ) v ( A ^ ( c(A) v B ) ) ) ) ) ) v ( A ^ ( c(A)
%     v ( ( ( c(A) ^ B ) v ( c(A) ^ c(B) ) ) v ( A ^ ( c(A) v B
%     ) ) ) ) ) ) ) ) ) != 1.
% end_of_list.

*************************

Symbol  0, relation, Dummy_symbol/0, atoms    1 --   1,          0.
Symbol  1, relation,      =/2, atoms    2 --   5,          1.
Symbol  2, function,      ^/3, atoms    6 --  13,          0.
Symbol  3, function,      v/3, atoms   14 --  21,          0.
Symbol  4, function,      c/2, atoms   22 --  25,          0.
Symbol  5, function,     d2/1, atoms   26 --  27,          0.
Symbol  6, function,      A/1, atoms   28 --  29,          0.
Symbol  7, function,     d3/1, atoms   30 --  31,          0.
Symbol  8, function,      B/1, atoms   32 --  33,          0.
Symbol  9, function,     d4/1, atoms   34 --  35,          0.
Symbol 10, function,     d5/1, atoms   36 --  37,          0.
Symbol 11, function,     d8/1, atoms   38 --  39,          0.
Symbol 12, function,     d9/1, atoms   40 --  41,          0.
Symbol 13, function,    d10/1, atoms   42 --  43,          0.
Symbol 14, function,    d11/1, atoms   44 --  45,          0.
Symbol 15, function,    d12/1, atoms   46 --  47,          0.
Symbol 16, function,    d13/1, atoms   48 --  49,          0.
Symbol 17, function,    d14/1, atoms   50 --  51,          0.
Symbol 18, function,    d16/1, atoms   52 --  53,          0.
Symbol 19, function,    d17/1, atoms   54 --  55,          0.
Symbol 20, function,    d18/1, atoms   56 --  57,          0.
Symbol 21, function,    d19/1, atoms   58 --  59,          0.
Symbol 22, function,    d20/1, atoms   60 --  61,          0.
Symbol 23, function,    d21/1, atoms   62 --  63,          0.
Symbol 24, function,    d22/1, atoms   64 --  65,          0.
Symbol 25, function,    d23/1, atoms   66 --  67,          0.

1. -^(v0,v1,v2) ^(v1,v0,v2)
2. -v(v0,v1,v2) v(v1,v0,v2)
3. -v(v0,v1,v2) -v(v3,v2,v4) -v(v3,v0,v5) v(v5,v1,v4)
4. -c(v0,v1) c(v1,v0)
5. -^(v0,v1,v2) v(v0,v2,v0)
6. -c(v0,v1) -c(v2,v3) -v(v3,v1,v4) -c(v4,v5) ^(v2,v0,v5)
7. ^(v0,v0,v0)
8. v(v0,v0,v0)
```

```
9. -c(v0,v1) v(v1,v0,1)
10. -c(v0,v1) ^(v1,v0,0)
11. v(1,v0,1)
12. v(v0,1,1)
13. ^(1,v0,v0)
14. ^(v0,1,v0)
15. ^(0,v0,0)
16. ^(v0,0,0)
17. v(0,v0,v0)
18. v(v0,0,v0)
19. -d2(v0) -A(v1) c(v1,v0)
20. -d3(v0) -d2(v1) -B(v2) v(v2,v1,v0)
21. -d4(v0) -d2(v1) -B(v2) ^(v2,v1,v0)
22. -d5(v0) -d3(v1) c(v1,v0)
23. -d8(v0) -B(v1) -c(v1,v2) -d2(v3) ^(v3,v2,v0)
24. -d9(v0) -A(v1) -d3(v2) ^(v2,v1,v0)
25. -d10(v0) -d4(v1) -d8(v2) v(v2,v1,v0)
26. -d11(v0) -d9(v1) -d10(v2) v(v2,v1,v0)
27. -d12(v0) -d2(v1) -d11(v2) ^(v2,v1,v0)
28. -d13(v0) -d2(v1) -d11(v2) v(v2,v1,v0)
29. -d14(v0) -d2(v1) -d11(v2) -c(v2,v3) ^(v3,v1,v0)
30. -d16(v0) -A(v1) -d13(v2) ^(v2,v1,v0)
31. -d17(v0) -d12(v1) -d14(v2) v(v2,v1,v0)
32. -d18(v0) -d16(v1) -d17(v2) v(v2,v1,v0)
33. -d19(v0) -d18(v1) -d5(v2) ^(v2,v1,v0)
34. -d20(v0) -d5(v1) -d18(v2) -c(v2,v3) ^(v3,v1,v0)
35. -d21(v0) -d5(v1) -d18(v2) v(v2,v1,v0)
36. -d22(v0) -d3(v1) -d21(v2) ^(v2,v1,v0)
37. -d23(v0) -d22(v1) -d19(v2) v(v2,v1,v0)
38. -d20(v0) -d23(v1) -v(v1,v0,1)
```

A.1.4 LCL168-1

```
%--------------------------------------------------------------------------
% File     : LCL168-1 : TPTP v3.1.1. Released v1.0.0.
% Domain   : Logic Calculi (Equivalential)
% Problem  : XEH is not a single axiom for the R-calculus
% Version  : [WW+90] axioms.
% English  : To show that XEH is not a single axiom, attempt to derive
%            from it any one of YQM, WO, XGJ or QYF, which are known
%            single axioms.

% Refs     : [WW+90] Wos et al. (1990), Automated Reasoning Contributes to
%          : [MW92]  McCune & Wos (1992), Experiments in Automated Deductio
% Source   : [WW+90]
% Names    : RC-2 [WW+90]

% Status   : Satisfiable
% Rating   : 0.80 v3.1.0, 0.86 v2.7.0, 0.80 v2.6.0, 0.75 v2.5.0, 0.83 v2.4.0, 1.00 v2.0.0
% Syntax   : Number of clauses     :    6 (   0 non-Horn;   5 unit;   5 RR)
%            Number of atoms       :    8 (   0 equality)
%            Maximal clause size   :    3 (   1 average)
%            Number of predicates  :    1 (   0 propositional; 1-1 arity)
%            Number of functors    :    4 (   3 constant; 0-2 arity)
%            Number of variables   :    5 (   0 singleton)
%            Maximal term depth    :    6 (   3 average)

% Comments : This is not how the problem is attacked in [WW+90].
%          : tptp2X -f otter:none:[set(auto),clear(print_given)] -t stdfof+add_equality:r LCL168-1.p
%--------------------------------------------------------------------------
set(prolog_style_variables).
set(tptp_eq).
.
.

list(usable).

% condensed_detachment, axiom.
-is_a_theorem(equivalent(X,Y)) |
-is_a_theorem(X) |
 is_a_theorem(Y).

% xeh, axiom.
 is_a_theorem(equivalent(X,equivalent(equivalent(Y,equivalent(equivalent(Y,Z),X)),Z))).
```

```
% try_prove_qyf, negated_conjecture.
-is_a_theorem(equivalent(equivalent(equivalent(a,b),equivalent(a,c)),equivalent(c,b))).

% try_prove_yqm, negated_conjecture.
-is_a_theorem(equivalent(equivalent(a,b),equivalent(equivalent(c,b),equivalent(c,a)))).

% try_prove_wo, negated_conjecture.
-is_a_theorem(equivalent(equivalent(a,equivalent(b,c)),equivalent(c,equivalent(b,a)))).

% try_prove_xgj, negated_conjecture.
-is_a_theorem(equivalent(a,equivalent(equivalent(b,equivalent(c,a)),equivalent(b,c)))).

end_of_list.

%------------------------------------------------------------------------------

********************

Symbol  0, relation, Dummy_symbol/0, atoms    1 --   1,          0.
Symbol  1, relation,  equal/2, atoms    2 --   5,          1.
Symbol  2, function, equivalent/3, atoms    6 --  13,          0.
Symbol  3, relation, is_a_theorem/1, atoms   14 --  15,          0.
Symbol  4, function,      c/1, atoms   16 --  17,          0.
Symbol  5, function,      b/1, atoms   18 --  19,          0.
Symbol  6, function,      a/1, atoms   20 --  21,          0.

1. -equivalent(v0,v1,v2) -is_a_theorem(v2) -is_a_theorem(v0) is_a_theorem(v1)
2. -equivalent(v0,v1,v2) -equivalent(v2,v3,v4) -equivalent(v0,v4,v5) -equivalent(v5,v1,v6)
-equivalent(v3,v6,v7) is_a_theorem(v7)
3. -equivalent(v0,v1,v2) -c(v0) -equivalent(v3,v0,v4) -b(v1) -a(v3) -equivalent(v3,v1,v5)
-equivalent(v5,v4,v6) -equivalent(v6,v2,v7) -is_a_theorem(v7)
4. -equivalent(v0,v1,v2) -c(v0) -equivalent(v0,v3,v4) -equivalent(v4,v2,v5) -b(v3) -a(v1)
-equivalent(v1,v3,v6) -equivalent(v6,v5,v7) -is_a_theorem(v7)
5. -equivalent(v0,v1,v2) -equivalent(v3,v2,v4) -c(v3) -b(v0) -equivalent(v0,v3,v5) -a(v1)
-equivalent(v1,v5,v6) -equivalent(v6,v4,v7) -is_a_theorem(v7)
6. -equivalent(v0,v1,v2) -c(v1) -equivalent(v1,v3,v4) -b(v0) -equivalent(v0,v4,v5)
-equivalent(v5,v2,v6) -a(v3) -equivalent(v3,v6,v7) -is_a_theorem(v7)
```

A.1.5 BOO061-1

```
%--------------------------------------------------------------------------
% File     : BOO061-1 : TPTP v3.1.1. Released v2.5.0.
% Domain   : Boolean Algebra
% Problem  : Single non-axiom M6D for Boolean algebra in the Sheffer stroke
% Version  : [EF+02] axioms.
% English  :

% Refs     : [EF+02] Ernst et al. (2002), More First-order Test Problems in
%          : [MV+02] McCune et al. (2002), Short Single Axioms for Boolean
% Source   : [EF+02]
% Names    : sheffer-mstar [EF+02]

% Status   : Satisfiable
% Rating   : 0.80 v3.1.0, 0.67 v2.7.0, 0.33 v2.6.0, 0.83 v2.5.0
% Syntax   : Number of clauses    :    3 (   0 non-Horn;   2 unit;   1 RR)
%            Number of literals   :    4 (   4 equality)
%            Maximal clause size  :    2 (   1 average)
%            Number of predicates :    1 (   0 propositional; 2-2 arity)
%            Number of functors   :    4 (   3 constant; 0-2 arity)
%            Number of variables  :    4 (   1 singleton)
%            Maximal term depth   :    5 (   2 average)

% Comments :
%          : tptp2X -f otter:none:[set(auto),clear(print_given)] -t stdfof+add_equality:r BOO061-1.p
%--------------------------------------------------------------------------
set(prolog_style_variables).
set(tptp_eq).


list(usable).

% reflexivity, axiom.
 equal(A,A).

% m6D, axiom.
 equal(nand(nand(A,nand(A,nand(B,B))),nand(B,nand(A,C))),B).

% prove_meredith_2_basis, negated_conjecture.
-equal(nand(nand(a,a),nand(b,a)),a) |
-equal(nand(a,nand(b,nand(a,c))),nand(nand(nand(c,b),b),a)).

end_of_list.
```

```
%-----------------------------------------------------------------------------

*************************

Symbol  0, relation, Dummy_symbol/0, atoms    1 --   1,          0.
Symbol  1, relation,  equal/2, atoms    2 --   5,          1.
Symbol  2, function,   nand/3, atoms    6 --  13,          0.
Symbol  3, function,      c/1, atoms   14 --  15,          0.
Symbol  4, function,      b/1, atoms   16 --  17,          0.
Symbol  5, function,      a/1, atoms   18 --  19,          0.

1.equal(v0,v0)
2.-nand(v0,v1,v2) -nand(v3,v2,v4) -nand(v3,v3,v5) -nand(v0,v5,v6) -nand(v0,v6,v7) nand(v7,v4,v3)
3.-nand(v0,v1,v2) -nand(v2,v1,v3) -nand(v3,v4,v5) -c(v0) -nand(v4,v0,v6) -nand(v1,v6,v7) -b(v1)
-nand(v1,v4,v8) -a(v4) -nand(v4,v4,v9) -nand(v9,v8,v4) -nand(v4,v7,v5)
```

A.1.6 ortholattice (needed by o1e1 and o1e4)

```
op(400, infix, [^,v]).  % infix operators

list(usable).

% Axioms for an ortholattice.

x ^ y = y ^ x.              % dependent on other axioms
% (x ^ y) ^ z = x ^ (y ^ z).  % dependent on other axioms

x v y = y v x.
(x v y) v z = x v (y v z).

c(c(x)) = x.
%  x v (y v c(y)) = y v c(y).  % follows from lemmas below
x v (x ^ y) = x.
x ^ y = c(c(x) v c(y)).

% Ortholattice lemmas.

x ^ x = x.
x v x = x.
c(x) v x = 1.
c(x) ^ x = 0.

1 v x = 1.
x v 1 = 1.
1 ^ x = x.
x ^ 1 = x.

0 ^ x = 0.
x ^ 0 = 0.
0 v x = x.
x v 0 = x.

end_of_list.
```

A.2 Problems in the SEQ Category

ALG008-1 ANA006-1 ANA006-2 BOO008-3 BOO019-1 BOO027-1 BOO030-1 BOO032-1 BOO033-1 BOO036-1 BOO037-1 BOO037-2 BOO037-3 BOO056-1 BOO057-1 BOO058-1 BOO059-1 BOO060-1 BOO061-1 CAT001-2 CAT002-2 CAT019-4 CAT019-5 CAT020-1 CAT020-2 CAT020-4 COL005-1 COL047-1 COL071-1 COL073-1 GEO078-4 GEO078-5 GEO078-6 GEO078-7 GEO157-1 GEO162-1 GEO163-1 GRP024-4 GRP025-2 GRP025-4 GRP026-2 GRP026-4 GRP027-1 GRP081-1 GRP112-1 GRP204-1 GRP207-1 GRP392-1 GRP393-1 GRP393-2 GRP394-1 GRP394-3 GRP395-1 GRP397-1 GRP398-1 GRP398-2 GRP398-3 GRP399-1 HEN013-1 HEN013-2 HEN013-3 HWC004-2 HWV038-1 LAT016-1 LAT024-1 LAT025-1 LAT046-1 LAT047-1 LAT048-1 LAT049-1 LAT050-1 LAT051-1 LAT052-1 LAT053-1 LAT054-1 LAT055-1 LAT055-2 LAT056-1 LAT057-1 LAT058-1 LAT059-1 LAT060-1 LAT061-1 LAT062-1 LAT063-1 LAT098-1 LAT100-1 LAT101-1 LAT102-1 LAT103-1 LAT104-1 LAT105-1 LAT109-1 LAT111-1 LAT113-1 LAT114-1 LAT115-1 LAT116-1 LAT119-1 LAT120-1 LAT121-1 LAT122-1 LAT126-1 LAT127-1 LAT128-1 LAT129-1 LAT130-1 LAT131-1 LAT132-1 LAT133-1 LAT134-1 LAT135-1 LAT136-1 LAT137-1 LCL136-1 LCL137-1 LCL142-1 LCL165-1 LCL267-3 LCL280-3 LCL288-3 LCL290-3 LCL291-3 LCL292-3 LCL338-3 LCL406-1 LCL407-1 LCL407-2 LCL408-1 LCL409-1 LCL410-1 LCL411-2 LCL412-1 LCL413-1 MGT038-2 NLP049-1 NLP050-1 NLP051-1 NLP052-1 NLP053-1 NLP180-1 NLP181-1 NLP182-1 NLP183-1 NLP184-1 NLP185-1 NLP186-1 NLP187-1 NLP188-1 NLP189-1 PUZ015-1 PUZ057-1 PUZ058-1 RNG007-5 RNG025-8 RNG031-6 RNG031-7 RNG042-1 RNG042-2 RNG042-3 RNG043-1 RNG043-2 ROB012-1 ROB012-2 ROB015-1 ROB028-1 ROB029-1 SWV021-1 SYN305-1

Bibliography

[1] "5th international planning competition, deterministic part," 2006. [Online]. Available: http://zeus.ing.unibs.it/ipc-5/

[2] A. Acharya and M. Tambe, "Collection oriented match," in *CIKM*, 1993, pp. 516–526.

[3] B. Aspvall, M. F. Plass, and R. E. Tarjan, "A linear-time algorithm for testing the truth of certain quantified boolean formulas," *Inf. Process. Lett.*, vol. 8, no. 3, pp. 121–123, 1979.

[4] R. Avnur and J. M. Hellerstein, "Eddies: continuously adaptive query processing," 2000, pp. 261–272. [Online]. Available: citeseer.ist.psu.edu/avnur00eddies.html

[5] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli, "Computing finite models by reduction to function-free clause logic," 2007.

[6] P. Baumgartner, A. Fuchs, and C. Tinelli, "Implementing the model evolution calculus," *International Journal on Artificial Intelligence Tools*, vol. 15, no. 1, pp. 21–52, 2006.

[7] ——, "Lemma learning in the model evolution calculus," in *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, ser. LNAI, M. Hermann and A. Voronkov, Eds., vol. 4246. Springer, 2006, pp. 572–586. [Online]. Available: ME-lemma-learning.pdf

[8] P. Baumgartner and C. Tinelli, "The Model Evolution Calculus," pp. 350–364. [Online]. Available: http://www.uni-koblenz.de/~peter/Publications/BaumgartnerTinelli-CADE-19.ps

[9] R. J. Bayardo, Jr. and R. Schrag, "Using csp look-back techniques to solve real-world sat instances," in *AAAI/IAAI*, 1997, pp. 203–208.

[10] C. E. Blair, R. G. Jeroslow, and J. K. Lowe, "Some results and experiments in programming techniques for propositional logic," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 633–645, 1986.

[11] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi, "An algorithm to evaluate quantified boolean formulae and its experimental evaluation," *J. Autom. Reasoning*, vol. 28, no. 2, pp. 101–142, 2002.

[12] Y. Chen, X. Zhao, and W. Zhang, "Long distance mutual exclusion for propositional planning," in *IJCAI*, 2007.

[13] K. Claessen and N. Sorensson, "New techniques that improve mace-style finite model finding," 2003, proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications.

[14] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi, "Benefits of bounded model checking at an industrial setting," in *Proc. of CAV*, ser. LNCS. Springer Verlag, 2001. [Online]. Available: http://www.mrg.dist.unige.it/~sim/simo/Publications/Data/bmcsat.ps.gz

[15] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[16] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, July 1962. [Online]. Available: http://www.acm.org/pubs/articles/journals/cacm/1962-5-7/p394-davis/p394-davis.pdf

[17] T. B. de la Tour, "Minimizing the number of clauses by renaming," in *CADE*, 1990, pp. 558–572.

[18] H. de Nivelle and J. Meng, "Geometric resolution: A proof procedure based on finite model search," in *Automated reasoning : Third International Joint Conference, IJCAR 2006*, ser. Lecture Notes in Artificial Intelligence, U. Furbach and N. Shankar, Eds., vol. 4130. Seattle, WA, USA: Springer, August 2006, pp. 303–317.

[19] H. Dixon, M. Ginsberg, D. Hofer, E. Luks, and A. Parkes, “Implementing a generalized version of resolution,” 2004, pp. 55–60.

[20] W. F. Dowling and J. H. Gallier, “Linear-time algorithms for testing the satisfiability of propositional horn formulae,” *J. Log. Program.*, vol. 1, no. 3, pp. 267–284, 1984.

[21] H. Fang and W. Ruml, “Complete local search for propositional satisfiability,” in *Proceedings of the Nineteenth National Conference in Artificial Intelligence (AAAI’04)*, 2004. [Online]. Available: citeseer.ist.psu.edu/fang04complete.html

[22] C. L. Forgy, “Rete: A fast algorithm for the many pattern/many object pattern match problem,” *Artificial Intelligence*, vol. 19, no. 1, 1982.

[23] H. Ganzinger and K. Korovin, “New directions in instantiation-based theorem proving,” in *Proc. 18th IEEE Symposium on Logic in Computer Science,(LICS’03)*. IEEE Computer Society Press, 2003, pp. 55–64.

[24] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book.* Prentice-Hall, 2002.

[25] J. Giarratano and G. Riley, *Expert Systems: Principles and Programming*, 4th ed. Course Technology, 2004.

[26] M. L. Ginsberg and A. J. Parkes, “Satisfiability algorithms and finite quantification,” in *KR2000: Principles of Knowledge Representation and Reasoning*, A. G. Cohn, F. Giunchiglia, and B. Selman, Eds. San Francisco: Morgan Kaufmann, 2000, pp. 690–701. [Online]. Available: citeseer.ist.psu.edu/ginsberg00satisfiability.html

[27] C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz, “Heavy-tailed phenomena in satisfiability and constraint satisfaction problems,” *Journal of Automated Reasoning*, vol. 24, no. 1/2, pp. 67–100, 2000. [Online]. Available: citeseer.ist.psu.edu/article/gomes99heavytailed.html

[28] C. P. Gomes, B. Selman, and H. A. Kautz, “Boosting combinatorial search through randomization,” in *AAAI/IAAI*, 1998, pp. 431–437.

[29] A. Gounaris, N. Paton, A. Fernandes, and R. Sakellariou, "Adaptive query processing: A survey," 2002. [Online]. Available: citeseer.ist.psu.edu/gounaris02adaptive.html

[30] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the satisfiability (sat) problem: A survey." *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, vol. 35, pp. 19–152, 1996.

[31] J. Gu and R. Puri, "Asynchronous circuit synthesis with boolean satisfiability." *IEEE Transactions on Computer-Aided Design*, vol. 14, pp. 961–973, August 1995.

[32] J. N. Hooker, "Generalized resolution and cutting planes," *Ann. Oper. Res.*, vol. 12, no. 1-4, pp. 217–239, 1988.

[33] H. H. Hoos and T. Stützle, "SATLIB: An Online Resource for Research on SAT," pp. 283–292. [Online]. Available: www.satlib.org

[34] *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99).* Stockholm, Sweden: Morgan Kaufmann, July 31-August 6 1999. [Online]. Available: http://www.dsv.su.se/ijcai-99/

[35] D. Jackson and M. Vaziri, "Finding bugs with a constraint solver," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 14–25, 2000.

[36] H. Kautz, B. Selman, and J. Hoffmann, "Satplan: Planning as satisfiability," in *Abstracts of the 5th International Planning Competition*. [Online]. Available: http://www.cs.rochester.edu/u/kautz/papers/kautz-satplan06.pdf

[37] H. A. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, 1992, pp. 359–363. [Online]. Available: http://www.cs.washington.edu/homes/kautz/papers/satplan.ps

[38] ——, "Unifying SAT-based and graph-based planning," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99).* Stockholm, Sweden: Morgan Kaufmann, July 31-August 6 1999, pp. 318–325. [Online]. Available: http://www.cs.washington.edu/homes/kautz/papers/ijcai99blackbox.ps

[39] D. Klahr, P. Langley, and R. Neches, *Production System Models of Learning and Development*. MIT Press, 1987.

[40] R. Letz and G. Stenz, “Dctp - a disconnection calculus theorem prover - system abstract,” in *IJCAR*, 2001, pp. 381–385.

[41] J. P. Marques-Silva, “The impact of branching heuristics in propositional satisfiability algorithms,” in *EPIA*, 1999, pp. 62–74.

[42] J. P. Marques-Silva and K. A. Sakallah, “GRASP - A New Search Algorithm for Satisfiability,” in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, November 1996, pp. 220–227. [Online]. Available: http://sat.inesc.pt/~jpms/research/papers/iccad96/iccad96.ps.gz

[43] W. McCune, “A davis-putnam program and its application to finite first-order model search: Quasigroup existence problem,” Technical Memorandum ANL/MCS-TM-194, Tech. Rep., 1994.

[44] W. McCune and L. Wos, “Otter - the CADE-13 competition incarnations,” *Journal of Automated Reasoning*, vol. 18, no. 2, pp. 211–220, 1997. [Online]. Available: citeseer.ist.psu.edu/article/mccune97otter.html

[45] D. G. Mitchell, B. Selman, and H. J. Levesque, “Hard and easy distributions of sat problems,” in *AAAI*, 1992, pp. 459–465.

[46] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr, “SETHEO and e-SETHEO - the CADE-13 systems,” *Journal of Automated Reasoning*, vol. 18, no. 2, pp. 237–246, 1997. [Online]. Available: citeseer.ist.psu.edu/moser97setheo.html

[47] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an Efficient SAT Solver,” in *Proceedings of the 38th Design Automation Conference (DAC’01)*, June 2001. [Online]. Available: http://www.ee.princeton.edu/~chaff/DAC2001v56.pdf

[48] D. A. Plaisted and S. Greenbaum, “A structure-preserving clause form translation,” *J. Symb. Comput.*, vol. 2, no. 3, pp. 293–304, 1986.

[49] A. Riazanov and A. Voronkov, "Splitting without backtracking," in *IJCAI*, 2001, pp. 611–617.

[50] ——, "The design and implementation of vampire," *AI Commun.*, vol. 15, no. 2-3, pp. 91–110, 2002.

[51] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J. ACM*, vol. 12, no. 1, pp. 23–41, 1965.

[52] P. S. Rosenbloom, J. E. Laird, and A. Newell, *The Soar Papers: Research on Integrated Intelligence*. MIT Press, 1993.

[53] M. G. Scutellà, "A note on dowling and gallier's top-down algorithm for propositional horn satisfiability," *J. Log. Program.*, vol. 8, no. 3, pp. 265–273, 1990.

[54] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD Conference*, 1979, pp. 23–34.

[55] B. Selman, H. A. Kautz, and B. Cohen, "Local search strategies for satisfiability testing," in *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, 1993. [Online]. Available: http://www.cs.cornell.edu/home/selman/papers-ftp/96.dimacs.walksat.ps

[56] B. Selman, H. J. Levesque, and D. G. Mitchell, "A new method for solving hard satisfiability problems," in *AAAI*, 1992, pp. 440–446.

[57] P. Singla and P. Domingos, "Memory-efficient inference in relational domains," in *AAAI*, 2006.

[58] D. Smith, J. Frank, and A. J'onsson, "Bridging the gap between planning and scheduling," 2000. [Online]. Available: citeseer.ist.psu.edu/smith00bridging.html

[59] P. Stephan, R. K. Brayton, and A. S. Vincentelli, "Combinational test generation using satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 15, pp. 1167–1176, Sept 1996.

[60] G. Sutcliffe, “The CADE-20 Automated Theorem Proving Competition,” *AI Communications*, vol. 19, no. 2, pp. 173–181, 2006. [Online]. Available: http://www.cs.miami.edu/~tptp/CASC/20/

[61] ——, “The 3rd ijcar automated theorem proving competition,” *AI Commun.*, vol. 20, no. 2, pp. 117–126, 2007.

[62] C. Thiffault, F. Bacchus, and T. Walsh, “Solving non-clausal formulas with dpll search,” in *CP*, 2004, pp. 663–678.

[63] G. Tseitin, “On the complexity of proofs in propositional logic,” *Seminars in Mathematics*, vol. 8, 1970.

[64] S. A. Wolfman and D. S. Weld, “The lpsat engine and its application to resource planning,” in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI’99)*. Stockholm, Sweden: Morgan Kaufmann, July 31-August 6 1999, pp. 310–316. [Online]. Available: http://www.dsv.su.se/ijcai-99/

[65] J. Zhang and H. Zhang, “Sem: a system for enumerating models,” in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI’95)*, Montréal, Québec, Canada, August 20–25 1995, pp. 298–303. [Online]. Available: http://www.ijcai.org/past/ijcai-95/

[66] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, “Efficient conflict driven learning in a Boolean satisfiability solver,” in *International Conference on Computer-Aided Design (ICCAD’01)*, Nov. 2001, pp. 279–285. [Online]. Available: http://www.ee.princeton.edu/~chaff/iccad2001_final.pdf