Solving Optimal Satisfiability Problems Through Clause-Directed A*

by

Robert J. Ragno

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by... ....
Brian C. Williams
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Solving Optimal Satisfiability Problems Through Clause-Directed A*

by

Robert J. Ragno

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering

Abstract

Real-world applications, such as diagnosis and embedded control, are increasingly being framed as *OpSAT* problems – problems of finding the best solution that satisfies a formula in propositional state logic. Previous methods, such as Conflict-directed A*, solve OpSAT problems through a weak coupling of A* search, used to generate optimal candidates, and a DPLL-based SAT solver, used to test feasibility. This paper achieves a substantial performance improvement by introducing a tightly coupled approach, *Clause-directed A* (ClA*)*. ClA* simultaneously directs the search towards assignments that are feasible and optimal. First, satisfiability is generalized to state logic by unifying the DPLL satisfiability procedure with forward checking. Second, optimal assignments are found by using A* to guide variable splitting within DPLL. Third, search is directed towards feasible regions of the state space by treating all clauses as *conflicts*, and by selecting only assignments that entail more clauses. Finally, ClA* climbs towards the optimum by using a variable ordering heuristic that emulates gradient search. Empirical experiments on real-world and randomly-generated instances demonstrate an order of magnitude increase in performance over Conflict-directed A*.

Thesis Supervisor: Brian C. Williams
Title: Associate Professor

Acknowledgments

I have received invaluable direction and inspiration from the people I worked with over the course of my research, both from lessons learned from previous work and from speculation about new approaches. I would like to thank Brian Williams, Seung Chung, John Van Eepoel, Samidh Chakrabati, and Ewa Dominowska. My research was supported in part by the NASA Cross Enterprise Technology Development program under contract NAG2-1466.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Within the Artificial Intelligence community, a significant amount of effort has been focused on the development and application of propositional satisfiability procedures. Performance in the last decade has increased by several orders of magnitude, both for local[22] and systematic, DPLL-style procedures[3, 4]. Through the transformation of problem descriptions, applications of satisfiability algorithms have expanded to include planning[15], verification[13, 14], test generation[24] and logic synthesis[16].

For many real-world tasks, however, simply finding feasible solutions is inadequate. Such tasks include most problems explored within the model-based reasoning community, such as diagnosis[8, 2], mode estimation[26, 17, 25], and embedded control problems[27]. These applications require the identification of *optimal*, feasible solutions. Optimality is crucial to the problem solutions in this domain: mode estimation, for example, determines the *most-likely* state of a system; mode reconfiguration determines the *least-cost* repair sequence.

These applications are commonly approached by framing them as *optimal satisfiability (OpSAT)* problems[28]. In these problems, a set of variables are constrained to values that collectively satisfy a set of propositional clauses and are mapped to a cost function that is to be minimized. To date, OpSAT problems, and more generally Optimal CSP's of all forms, have been solved by layering a best-first candidate generator on top of a procedure that tests feasibility. The candidate generator guides the selection of variable assignments towards configurations that have minimal cost.

The test procedure guides the selection of variable assignments towards those that satisfy the constraints.

Progress has been made in developing high-performance OpSAT solvers with techniques, such as Conflict-directed A*[28], which bias the generator slightly towards assignments that are feasible as well as optimal. To speed up the search, Conflict-directed A* produces inconsistent sets of assignments, called *conflicts*, during the test phase. In the generate phase, the selection of assignments is guided away from these conflicts. In this manner, Conflict-directed A* moves from a two-phase model towards guiding the search at every step to assignments that are *both* feasible and optimal. However, the algorithm still has a distinct division between the optimizing search component and the feasibility testing component.

This thesis introduces *Clause-directed A**, a method of solving OpSAT problems which fully merges the two components. It does so by unifying the DPLL satisfiability procedure with the variation on A* employed in Conflict-directed A*. Clause-directed A* thus realizes the goal of directing the search at every step towards optimality and feasibility, simultaneously. This produces an efficiency boost by reducing searching.

Basic techniques of optimal search and satisfaction solving are first presented as a basis. Conflict-directed A* is then covered both as a background in OpSAT techniques and issues and as a starting point for the development of Clause-directed A*. A set of enhancements to the basic satisfaction algorithm then build up to the total Clause-directed A* algorithm. Finally, the performance gains are demonstrated by examining the performance on randomized problems.

Chapter 2

Optimization and Satisfiability

2.1 The OpSAT Problem

An optimal satisfiability problem is of the form:

$$\min f(\mathbf{x}) s.t. \mathbf{C}(\mathbf{x})$$

$x_i \in \mathbf{x}$ is a variable with domain $D(x_i)$. $C(\mathbf{x})$ is a set of clauses expressed in propositional state logic. In state logic, each proposition is an assignment, $x_i = v_{ij}$, where $x_i \in \mathbf{x}$ and $v_{ij} \in D(x_i)$. Propositions are composed into formula using the standard logical connectives - and ($\wedge$), or ($\vee$) and not ($\neg$) - and are reduced to clauses using a standard DNF procedure.

The cost function $f(\mathbf{x})$ is specified by a set of attribute cost functions, $f_i : D(x_i) \rightarrow \Re$, which specify the cost of individual variable assignments, and a function F, which combines attribute costs into a global cost. For simplicity of presentation, we assume that F is addition (+). [1]

We demonstrate clause-directed A* using a simple example, whose variables and respective domains are listed below. Each domain value v is paired with its cost c

[1] [28] extends the algorithms to OpSAT problems with general cost functions that satisfy a property of mutual preferential independence.

(i.e., $v : c$).:

$$\begin{aligned} A: &\quad \{x:3, y:0, z:2\} \\ B: &\quad \{x:0, y:2, z:1\} \\ C: &\quad \{x:0, y:0, z:0\} \\ D: &\quad \{x:0, y:0\} \end{aligned}$$

$C(\mathbf{x})$ for the problem consists of five clauses:

$$\begin{aligned} &(A = y) \vee (B = x) \vee \neg(D = y) \\ &(B = y) \vee (C = y) \\ &(A = z) \vee (C = y) \\ &\neg(C = y) \vee \neg(A = y) \vee (C = x) \\ &\neg(B = x) \end{aligned}$$

The solution to this OpSAT problem is:

$$\{A = z, B = z, C = y, D = x\}$$

2.2 Optimal search with A*

The first capability needed to solve an OpSAT problem is optimal search. While there is no ideal method for all cases, A* has become accepted as the leading general search method that is guaranteed to produce an optimal result. A* is appealing for both its simplicity and its efficiency. A* search finds a minimal-cost path to a goal by performing a best-first search based on an objective function f of the search nodes, where f is the sum of the cost of the partial path to that node, g, and a local heuristic estimate for the distance from that node to a goal, h.

The heuristic must be *admissible* to ensure optimality: the heuristic value at any given node must be an underestimate of the cost of the partial path from that node to the goal. Although not necessary for correctness of the result, A* also specifies that dynamic programming is used: nodes that are reached through multiple paths should only be reached through the best-cost path. (Also pursuing worse-cost paths would be inefficient in both time and space, since they cannot be part of the optimal answer).

Optimality is ensured because of the best-first search. When a solution is produced, if any better-cost paths existed, they would have been found earlier. The cost for any of the partial paths still under consideration can only be *less* than the true cost, so they could never gain a lower cost estimate through extension than the node produced. Additionally, the cost estimate of the solution node must have converged to the true cost, since the only underestimate possible at that point is zero.

2.3 Satisfiability and DPLL

To develop an OpSAT search method that is optimal, we start with a complete satisfiability procedure. Most modern, complete procedures[19, 18, 20, 1, 29], build upon the venerable, Davis, Putnam, Logemann, Loveland (DPLL) algorithm[3]. DPLL is a systematic search procedure, that is, it does not revisit states. In addition, DPLL is able to check unsatisfiability conditions, which is useful for knowledge compilation and embedded control tasks[27]. Given these features, DPLL offers an ideal starting point for our unification.

```
function DPLL(theory)
  returns true if a solution exists, false otherwise.
  unit-propagate(theory)
  if theory is empty
    then return true
  prop ← Select-Unassigned-Proposition(theory)
  if DPLL(theory ∪ {prop})
    then return true
  if DPLL(theory ∪ {¬prop})
    then return true
  return false
```

Figure 2-1: Recursive definition of DPLL satisfiability.

The DPLL procedure, shown in Figure 2-1, performs depth-first, backtrack search, with unit propagation and arbitrary splitting on boolean variables. DPLL takes as input a CNF theory, consisting of a set of disjunctive clauses. Unit propagation is based on the unit resolution rule, which states that, if all literals of a clause are assigned false, save one, then the remaining literal must be true. Unit propagation computes the closure of this rule. During unit propagation, a clause is entailed if at least one literal is assigned true, in which case the clause is removed. A clause is shortened if a literal is assigned false, by dropping the literal. A theory is inconsistent if a clause exists that has no remaining literals. The theory is proven to be satisfiable, as soon as all clauses are entailed by the current assignment.

If unit propagation completes with at least one entailed clause, DPLL splits on an unassigned proposition, first assigning it *true* and continuing the search. If the subsearch succeeds, then the theory is satisfiable. If not, the same proposition is assigned *false*, and the subsearch is repeated. If both assignments fail, the theory is inconsistent.

Chapter 3

Review of Conflict-directed A*

Conflict-directed A* is a technique that was developed to address OpSAT problems[28] in the context of model-based autonomy. It is a generate-and-test algorithm that alternates between best-first generation of candidates and testing the consistency of these candidates. This is made efficient through the use of conflicts.

The search is performed over partial sets of assignments to the decision variables, y. A node contains a partial set of assignments to the variables, and the search tree initially starts out with a single empty node. Each partial assignment corresponds to a region of the state space of total assignments (to all of the problem variables) – the initial empty set of assignments corresponds to the entire state space.

The cost of a node n, $f(n)$, is calculated as simply the sum of the costs of all of the assignments in the partial set of assignments, $g(n)$, plus the sum of the costs of the least-cost assignment for each of the variables that is not represented, $h(n)$. The meaning of g varies depending on the application. In model-based autonomy, for example, these assignment costs may represent the cost to transition a component to a particular mode. The goal of the search could then be mode reconfiguration, an attempt to find the least-cost configuration that satisfies the given model and goal constraints. For mode estimation, the cost of each assignment could be the negative logarithm of the probability that the component (variable) is in that mode (assignment). Thus, the least-cost node would be the set of assignments with the highest probability of occurring, if the probabilities are assumed to be independent (since addition over the logarithms is analogous to multiplication over the probabilities).

The value of $h(n)$ gives a best-case estimate for all possible extensions of n, since every variable must have some assignment. This makes h an underestimate, and it is admissible for A* search.

is to second factor is an underestimate and is an admissible estimate for A* search.

The testing is done by a satisfaction algorithm, or satisfaction engine. While many techniques have been developed for this purpose, a common and simple one is DPLL. However, it is important to note that any algorithm that implements a satisfiability solver could be used (subject to the constraint that it be able to generate conflicts, as described below). This allows for the flexibility to use more advanced engines or algorithms that are appropriate for a particular domain.

If a candidate set of assignments is not consistent with the logical theory, the satisfaction engine must produce a *conflict* for that candidate. A conflict is defined as a set of assignments that are collectively inconsistent with the theory. This conflict must be implied by the candidate; in other words, it must explain why the candidate is inconsistent. While the candidate itself would always be a valid conflict, a more minimal conjunction of assignments is preferable, since it specifies a broader range of inconsistent states.

As the name implies, the search is a variation on an A* search. At each step, the lowest-cost node in the search queue is removed and tested to see if it is consistent with the theory. This is first checked by inspecting whether it entails a violation of each of the conflicts. If all of the conflicts are resolved, the partial set of assignments is passed to the sat engine as a candidate. If the satisfaction engine verifies it as consistent with the theory, the search has generated a feasible solution. Because of the best-first property of A*, this solution is also guaranteed to be of lower cost than that of any node remaining in the search tree (or any extension to such nodes). If the candidate is not consistent, the satisfaction engine generates a conflict, which is added to the list of known conflicts.

The first conflict that is not resolved is used to expand the parent node. A conflict implicitly specifies a set of assignments that would each independently violate it.

Each of these that is consistent with the parent partial set of assignments generates a new node whose partial assignments are the union of those of the parent and that assignment. These children nodes are placed on the search queue, and the process is repeated.

Chapter 4

Clause-directed A*

The remainder of this paper derives Clause-directed A* by introducing a series of augmentations to DPLL which directs it to steeply climb towards optimal assignments and assignments that entail the clauses of the theory.

4.1 Filtering Domains by Unit Propagation

Techniques used to solve constraint satisfaction problems exploit variable domain knowledge. This is central to filtering methods, such as forward checking and constraint propagation for arc consistency. This knowledge is also used to implement variable ordering heuristics. DPLL is designed to solve problems phrased in propositional logic, where each proposition can take on a boolean value – either *true* or *false*. Because of this, CSP methods that take advantage of domain knowledge are not useful for satisfiability search. Since there are only two possible values for any proposition, a domain reduction is equivalent to either an assignment or an inconsistency. In addition, the unassigned propositions all have domains of the same size, so no comparison can be made.

State logic makes variable domains explicit. Propositions refer to variable-value assignments, and clause literals refer to either asserting or refuting these assignments. Clause-directed A* explicitly represents the reduced domains caused by unit propagation, replacing the implied constraints of mutual exclusion and domain exhaustion.

Unit propagation is carried out on the reduced domains as well as the clauses. Propagating a positive assignment requires reducing the corresponding variable domain to a singleton; propagating a negative assignment requires removing the corresponding value from its domain. A domain that is reduced to a singleton represents a unit clause and the associated proposition must be propagated through the clauses. A domain restricted by the propagation of a positive literal must also propagate the negative literal corresponding to each eliminated assignment in the domain. A domain reduced to an empty set represents an inconsistency.

The result of this treatment of state logic is to perform the methods of filtering used in CSP algorithms with the extended unit propagation. Performing this unit propagation is analogous to propagating constraints until full arc consistency is achieved. Similarly, each arbitrary truth assignment made by the DPLL algorithm translates into eliminating a value from a domain or reducing a domain to a single value. The combination of branching and unit propagation is demonstrated on the example in figure 4-1.

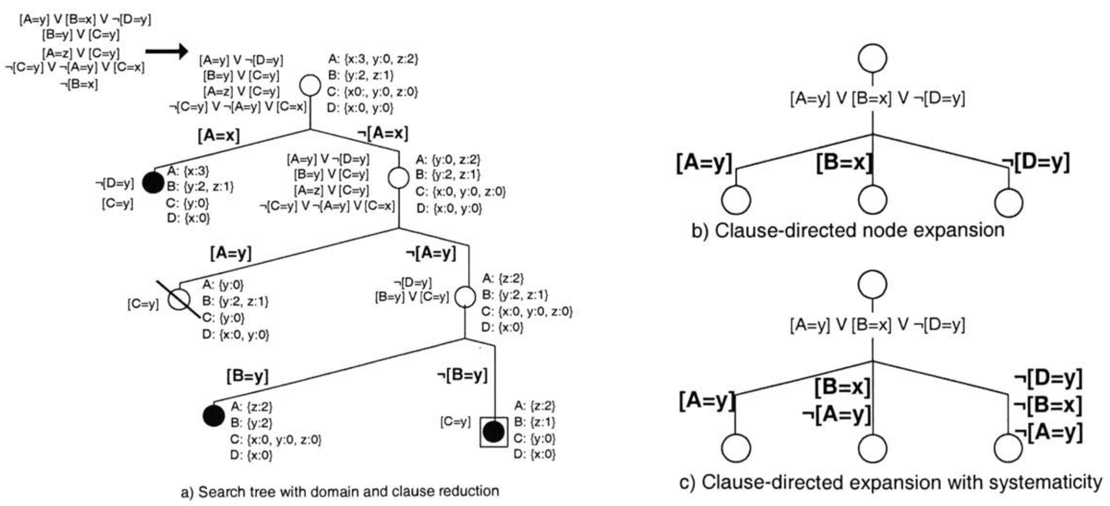

Figure 4-1: Search tree for DPLL with variable domain filtering (left), and details of node expansion (right). Branches are labeled with selected assignments or their negations. Tree nodes are labeled with reduced clauses to the left and reduced variable domains to the right. Nodes with inconsistent assignments are crossed out, and nodes that entail all clauses are filled in. The optimal, feasible node is indicated with a box.

4.2 Incorporating Optimal Search

```
function DPLL-A*(init-theory, init-varset)
  returns the minimal cost solution.
  unit-propagate(init-theory, init-varset)
  initial-node ← {init-theory, init-varset }
  queue ← {initial-node }
  loop do
      if queue is empty
        then return nil
      best-node ← Pop-First-Element(queue)
      if theory(best-node) is empty
        then return best-node
      theory ← theory(best-node)
      varset ← varset(best-node)
      next-var ← First-Element(varset)
      assign ← First-Element(valset(next-var))
      pos-child ← {theory ∪ {assign}, varset}
      neg-child ← {theory ∪ {¬assign}, varset}
      for each child in {pos-child, neg-child} do
        unit-propagate(theory(child), varset(child))
        if not Conflict?(theory(child), varset(child))
          then Insert-By-Cost(queue, child)
      end
  end
```

Figure 4-2: Optimal satisfiability using DPLL-A*.

Next the procedure is extended so that at each step it selects assignments that move towards an optimal solution, as well as towards feasible solutions. The extension is based on A* search, which complements the completeness of DPLL with a guarantee that the solution is optimal.

To review, A* search finds a minimum-cost path to a goal by expanding nodes in best-first order using a function f on nodes, where f is the sum of the cost g of the partial path to that node, and a local heuristic estimate h for the distance from that node to a goal.

An algorithm, DPLL-A*, that merges DPLL and A* search is shown in Figure 4-2. DPLL-A* replaces the recursive definition of DPLL with one that maintains an explicit stack discipline, similar to A*, and dequeues nodes in best-first order according to f. Each branch of the search tree is labeled with an assignment, and the cost of that branch is the assignment cost, $f_i(v_{ij})$ for $x_i = v_{ij}$. The cost g of a node is the sum of the assignment costs from the root to that node. The heuristic cost h is the minimum cost to complete the assignment, to unassigned variables. The h used is the sum of the cost, for each unassigned variable, of the least-cost value in its reduced domain. It follows that h is admissible, from the property of mutual preferential independence, as discussed in [28].

When a node is expanded, unit propagation is performed on each child in order to prune inconsistent children, and to reduce the domains of its unassigned variables. Reducing the domains of a child before enqueuing the child has the effect of improving its heuristic estimate h. The tradeoff is that a cost of unit propagation is incurred. This is an unnecessary cost if the child is never dequeued.

The results of DPLL-A* for our simple example is shown in Figure 4-1a. The algorithm walks consecutively along the right branches towards the optimal solution, indicated by a box. The reduced domains for each node, resulting from unit propagation, are shown to the right. There is a significant reduction in domain size at each step.

4.3 Using Clauses to Prune Infeasible Subspaces

A key insight underlying Conflict-directed A* and its early ancestor, GDE[7], is that conflicts can be used to effectively prune infeasible subspaces, at each step in the search. This is accomplished by expanding a search node at each step with only those assignments that resolve an unresolved conflict. In Conflict-directed A*, variables are divided into decision variables, which appear in f, and non-decision variables, which do not. The conflicts only guide assignments to decision variables. The remaining variables are assigned within the standard DPLL search.

Clause-directed A* (ClA*) pushes conflict-direction to the extreme, based on the insight that *every clause* may be viewed as a conflict, by identifying subsets of the

```
function Clause-directed-A*(init-theory, init-varset)
  returns the minimal cost solution.
  unit-propagate(init-theory, init-varset)
  initial-node ← {init-theory, init-varset }
  queue ← {initial-node }
  loop do
        if queue is empty
          then return nil
        best-node ← Pop-First-Element(queue)
        if theory(best-node) is empty
          then return best-node
        theory ← theory(best-node)
        varset ← varset(best-node)
        con ← Pop-First-Element(theory(best-node))
        for each literal in con do
          child ← {theory ∪ {literal}, varset}
          unit-propagate(theory(child), varset(child))
          if not Conflict?(theory(child), varset(child))
            then Insert-By-Cost(queue, child)
        end
  end
```

Figure 4-3: Algorithm with unit propagation, DPLL-style arbitrary choice, A*, and clause-directed expansion.

infeasible space to be pruned. ClA* is shown in Figure 4-3. At every search step, ClA* selects a clause that is not entailed by the current search tree node, and expands the node with only those assignments that entail this clause, as depicted in Figure 4-1b. More specifically each literal contained in the selected clause is used to form a child node. The reduced domains of each child is created by filtering the domains of its parent, according to the newly selected literal and unit propagation. This clause-directed approach strongly biases the search towards feasible subspaces.

Note that at each step, node expansion has several unentailed clauses that it may choose from. ClA* selects the shortest clause, since this eliminates the largest portion of the infeasible space.

This expansion technique is similar to clause-based branching rules used in some SAT algorithms [12], although the ClA* search is not depth-first. It is important to note that using the clauses in this manner does not affect the optimality of the result, only the efficiency of the search.

4.4 Terminating with Feasible Subspaces

A second key insight introduced by the GDE system [7], is that the space of solutions that satisfy a theory can be represented compactly by a set of partial assignments, called prime implicants, that entail that theory. Typically the set of prime implicants is dramatically smaller in size, compared to the set of explicit solutions.

If a search algorithm can identify that a partial assignment is an implicant of constraints $C(\mathbf{x})$, then it can stop without expanding the partial assignments further, since all extensions of it are guaranteed to be solutions. This can dramatically reduce the size of the search tree explored.

This feature of GDE depended on having a complete list of conflicts for a theory. It needed to be dropped by successors, such as Conflict-directed A*, since the loose coupling between generation and test prevented the search algorithm from determining entailment of all constraints. However, through its tight coupling, Clause-directed A* is able to easily identify implicants, by simply detecting when all clauses of a search node have been eliminated. At this point the search procedure can return the node's partial assignment as a prime implicant, without further expansion.

4.5 Restoring Systematicity

```
function Clause-directed-A*(init-theory, init-varset)
  returns the minimal cost solution.
  unit-propagate(init-theory, init-varset)
  initial-node ← {init-theory, init-varset }
  queue ← {initial-node }
  loop do
       if queue is empty
         then return nil
       best-node ← Pop-First-Element(queue)
       if theory(best-node) is empty
         then return best-node
       theory ← theory(best-node)
       varset ← varset(best-node)
       con ← Pop-First-Element(theory(best-node))
       siblings ← { }
       for each literal in con do
         child ← {theory ∪ {literal ∪ siblings}, varset}
         unit-propagate(theory(child), varset(child))
         if not Conflict?(theory(child), varset(child))
           then Insert-By-Cost(queue, child)
         siblings ← siblings ∪ {¬literal}
       end
  end
```

Figure 4-4: Clause-directed A* augmented with systematic search.

DPLL and DPLL-A* have the property that they are systematic, that is, they do not visit a search state twice. This property is lost with the clause-directed approach as presented above. The issue is that two children of a node may represent sets of total assignments that are overlapping. This can lead to wasted effort.

We restore stematicity by explicitly disallowing any overlap between a child and its siblings. In order of expansion, each child asserts, not only its corresponding literal, but also the negation of each literal for each of its preceding siblings. This is depicted in Figure 4-1c. Thus, the assignments that would normally lie within the intersection between two children, are now contained by the child that appears the earliest in the ordering. The updated algorithm is shown in Figure 4-4.

4.6 Dynamic Ordering Biased Towards Optimality

Search efficiency is significantly impacted by the order in which the clauses are selected, and the order in which the corresponding literals are expanded. This ordering can be decided using heuristics developed for SAT and CSP solvers, such as covering shorter clauses first, and expanding more constrained variables first. These heuristics, however, are designed only to bias the search towards feasible solutions, not optimal solutions. Heuristics should also bias selection away from subtrees that are most likely to contain nodes that have costs significantly worse than the current best node.

I have explored one such heuristic, which has proven empirically to be quite effective, and which at an intuitive level emulates characteristics of steepest, gradient descent. In particular, given two choices of clauses, the heuristic selects the clause containing two children with very different cost. This clause is most likely to minimize exploration of the search tree, since it is likely that the child with the higher cost will never be searched or will only be searched to a shallow degree.

4.7 Comparison to Conflict-Directed A*

In many ways, the resulting Clause-directed A* search is a modification and refinement of the earlier Conflict-directed A*. That system separates the solution process into two components – generating candidates in a best-first manner, and testing the feasibility of these candidates using a SAT solver. The satisfiability engine produces conflicts for each infeasible candidate, which is used to guide the generating search away from the infeasible subspace.

While this approach is flexible and modular, it contains a significant inefficiency. The SAT solver is called repeatedly and is run to completion each time. Clause-directed A* tries to address this by combining the satisfiability and optimization searches into one process. The expansion of the search tree is guided directly by the clauses of the constraining theory, which directs the search away from the infeasible subspace.

Conflict-directed A* does handle one aspect of OpSAT problems in a more straightforward manner than Clause-directed A* does. OpSAT problems allow both *decision* variables, which are searched over to form the optimal-cost assignment, and *state* variables, which can appear in the logical theory but have no cost and are not mentioned in the solution. For Clause-directed A*, these *state* variables can be represented simply as ordinary variables with zero cost for every assignment. This follows naturally from the concept that Clause-directed A* folds together the satisfiability search and the optimizing search. These *state* variables can be removed from the partial assignments in any generated solution, leaving only the *decision* variable assignments that OpSAT requests.

Chapter 5

Experimental Results

Comparing the performance of Clause-directed A* relative to Conflict-directed A* evaluates the benefit of a tight coupling at each search step as opposed to the loose coupling of generate and test. Clause-directed A* is implemented as an extension to Conflict-directed A*, with most of the base code shared. As a SAT solver, Conflict-directed A* uses an implementation of DPLL that performs incremental unit propagation using a truth maintenance system. Performance was compared for both randomly-generated problem instances and for subsystems of NASA and Air Force spacecraft. The statistics on randomized instances are presented, since they offer a larger sample space to draw from. The smaller set of real world examples show similar performance, although more testing is needed.

Problem instances were parameterized according to total number of variables, number of decision variables (variables with non-zero costs), number of clauses, maximum variable domain size, and maximum clause length. For each parameter setting, problem instances were generated with a uniform distribution for the domain and clause sizes and for the cost distribution for each variable. Statistics were taken for at least 1,000 problem instances per parameter setting. The results are shown in Table 5.1.

For most parameter values, Clause-directed A* offers a factor of at least ten speedup over Conflict-directed A*. The performance margin is reduced in general when most of the variables are decision variables (i.e., have non-zero cost). In this

Problem Parameters					Conflict-Directed	Clause-Directed
Variables	Max Dom. Size	Decision Vars	Theory Clauses	Max Clause Len	Avg Solv. Time	Avg Solv. Time
150	5	40	150	10	0.18	2.1
150	5	75	150	10	0.21	1.4
150	5	110	150	10	1.2	1.3
100	4	15	40	5	0.15	1.8
100	4	50	40	5	0.18	1.3
100	4	85	40	5	0.16	0.8

Table 5.1: Sample of empirical results on randomized problems: Solving time for Conflict-directed A* and Clause-directed A*.

case, the problems are strongly shifted from traditional satisfiability, towards A* search. The fluctuating performance gains suggest that there may be further opportunities to merge insights from each of the two algorithms.

In addition, experimental results in [28] demonstrate that Conflict-directed A* strongly dominates Constraint-based A*, an approach that considers only optimality during generation, not feasibility. This comparison can be extended to the Clause-directed A* case to examine the abstract growth of each algorithm, based on the number of nodes explored and not on the time it takes to complete. As Table 5.2 shows, Clause-directed A* edges out Conflict-directed A*.

Together these experiments argue for the power of directing search towards conflicts or clauses that prune infeasible subspaces. The results also imply that the simplicity of Clause-reducing A* reduces the algorithm overhead enough to noticeably increase the performance. This can be most clearly seen by the performance on small problems, where Clause-directed A* only marginally reduces the time and space requirements in terms of nodes examined but still significantly outperforms Conflict-directed A* in terms of running time.

Problem Parameters					BFS		Conflict-Directed		Clause-Directed	
Vars	Max Dom.	Dec. Vars	Constr.	Max Length	BFS expand	BFS size	CDA* expand	CDA* size	ClA* expand	ClA* size
30	10	20	10	6	929	4060	6	27.3	5.1	13.1
30	10	20	30	6	1430	5130	9.71	94.4	8.3	33.2
30	10	20	50	6	3790	13400	5.75	16	4.9	17.3
30	5	20	10	5	683	1230	3.33	6.33	3.2	7.8
30	5	20	30	5	2360	3490	8.13	17.9	6.0	17.9
30	5	20	50	5	4270	6260	12	41.3	7.3	21.7
30	5	10	10	5	109	149	4.2	7.2	3.6	5.5
30	5	10	30	5	149	197	5.4	7.2	4.2	6.7
30	5	10	50	5	333	434	6.4	9.2	6.1	7.2

Table 5.2: Sample of empirical results on randomized problems: Nodes explored and maximum in search at once for simple BFS, CDA* and ClA*.

Chapter 6

Conclusion

Optimal satisfaction, as found in the OpSAT class of problems, is crucial for a number of important real-world applications, including diagnosis, mode estimation, and mode reconfiguration. Other problems, such as planning tasks, are also being translated into optimal satisfaction problems in order to leverage the more general techniques. This has a particularly strong impact on the field of autonomous reasoning, where such problem domains naturally occur.

This thesis presented a method for solving these problems efficiently. While based on previous work in the area, Clause-directed A* distinguishes itself by both its simplicity and its high performance. It achieves both of these by tightly coupling several powerful concepts of Artificial Intelligence.

Clause-directed A* is built as a combination of four techniques that guide the search towards the optimal solution while avoiding infeasible assignment sets as early as possible. First, Clause-directed A* uses unit propagation to prune the variable domains and improves the bias towards feasible solutions. Second, Clause-directed A* guides variable splitting with an A* search variant designed for constraint problems. Third, Clause-directed A* prunes infeasible subspaces by treating every clause as a conflict, reducing wasted search time. Fourth, Clause-directed A* directing the search toward partial assignments corresponding to feasible subspaces, an analogous technique to the use of kernel diagnoses in model-based diagnosis[8, 5].

The experiments on randomly-generated problems are promising. Clause-directed

A* has a clear advantage in some types of problems over Conflict-directed A*, which already performs much better than a traditional generate-and test search. Preliminary results with real-world problems suggest similar advantages in performance.

There is much room for improvement and investigation. Besides the exploration of performance ratios with various types of problems, the transition between satisfaction problems and optimization problems can provide much useful data. Since very efficient techniques are known for each extreme case, the performance of Clause-directed A* should approach that of the simpler techniques on which it is based. Given that, there is also the possibility of incorporating more advanced methods of satisfaction solving and optimal search into the construction of the synthesized algorithm. Some aspects of the given development also present the opportunity for testing and improvement as well, such as the choice of heuristics for ordering the clauses.

Clause-directed A* has shown itself to be a simple and efficient way to attack the complex problem of finding optimal solutions to valued satisfaction problems. Additionally, is able to be tuned for specific problems and improved heuristics because of its increased flexibility and generality of techniques over the methods it is based on.

Bibliography

[1] R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI-97*, pages 203–208, 1997.

[2] S. Chung, J. Van Eepoel, and B. C. Williams. Improving model-based mode estimation through offline compilation. In *Proceedings of ISAIRAS-01*, 2001.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[4] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[5] J. de Kleer, A. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

[6] J. de Kleer and B. C. Williams. Back to backtracking: Controlling the ATMS. In *Proceedings of AAAI-86*, pages 910–917, 1986.

[7] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[8] J. de Kleer and B. C. Williams. Diagnosis with behavioral modes. In *Proc IJCAI-89*, pages 1324–1330, 1989.

[9] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[10] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[11] W. Hamscher, L. Console, and J. de Kleer. *Readings in Model-Based Diagnosis.* Morgan Kaufmann, San Mateo, CA, 1992.

[12] J.N. Hooker and V. Vinay. Branching rules for satisfiability. *Automated Reasoning*, 15, 1995.

[13] D. Jackson, S. Jha, and C. Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Trans. Progamming Languages and Systems*, 20(2), March 1998.

[14] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the ally constraint analyzer. In *Proc. Intl. Conf. Software Engineering*, Limerick, Ireland, 2000.

[15] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc AAAI-96*, pages 1194–1201, 1996.

[16] W. Kunz and D. Sotoffel. *Reasoning in Boolean Networks.* Kluwer Academic Publishers, 1997.

[17] A. Malik and P. Struss. Diagnosis of dynamic systems does not necessarily require simulation. In *Proceedings DX-97*, 1997.

[18] J. Pl Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[19] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Electronic Notes in Discrete Mathematics*, 9, 2001.

[20] P. Nayak and B. C. Williams. Fast context switching in real-time propositional reasoning. In *Proceedings of AAAI-97*, pages 50–56, 1997.

[21] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.

[22] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, pages 440–446, 1992.

[23] R. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[24] P. Stephan, S. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15:1167–1176, 1996.

[25] B. C. Williams, S. Chung, and V. Gupta. Mode estimation of model-based programs: Monitoring systems with complex behavior. In *Proceedings of IJCAI-01*, 2001.

[26] B. C. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proc AAAI-96*, pages 971–978, 1996.

[27] B. C. Williams and P. Nayak. A reactive planner for a model-based executive. In *Proceedings of IJCAI-97*, 1997.

[28] B. C. Williams and R. J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *to appear, Journal of Discrete Applied Math*, 2002.

[29] H. Zhang. SATO: an efficient propositional prover. In *Proc. Intl. Conf. Automated Deduction*, pages 272 – 275, 1997.