A Partial Instantiation based First Order Theorem Prover

V. Chandru* I.I.Sc.
J.N. Hooker† C.M.U.
G. Rago‡ U. of Pisa
A. Shrivastava§ Sanchez C.A.

Abstract

Satisfiability algorithms for propositional logic have improved enormously in recent years. This increases the attractiveness of satisfiability methods for first order logic that reduce the problem to a series of ground-level satisfiability problems. Partial Instantiation for first order satisfiability differs radically from standard resolution based methods. Two approaches to partial instantiation based first order theorem provers have been studied by R. Jeroslow [10] and by Plaisted and Zhu [14]. Hooker and Rago [8, 9] have described improvements of Jeroslow's approach by a) extending it to logic with functions, b) accelerating it through use of satisfiers, as introduced by Gallo and Rago [6] and c) simplifying it to obtain further speedup. The correctness of the Partial Instantiation algorithms described here for full first-order logic with functions as well as termination on unsatisfiable formulas are shown in [9]. This paper describes the implementation of a theorem prover based on the primal algorithm and its application to solving planning problems. We obtained improved efficiency by incorporating incrementality into the primal algorithm (incremental blockage testing). This extended abstract describes the Partial Primal Instantiation algorithm, its implementation and preliminary results on first order formulation of planning problems.

1 Algorithm PPI (Primal Partial Instantiation)

Variants: Two formulas are said to be variants of each other when they can be unified by renaming substitutions.
Satisfier Mapping: A mapping, S, that associates with each clause C in F, a literal $S(C)$ in C, is called a satisfier mapping for F if, for some truth valuation v, v makes $S(C)$ true for every clause C. We refer to $S(C)$ as the satisfier of C. It is a true satisfier if $S(C)$ is an atom, and false otherwise.
Blockage: Given a satisfier mapping S for a quantifier-free formula F, a pair of satisfiers $P(t), P(t')$ are blocked if:

1. $P(t)$ is a true satisfier;
2. $P(t')$ is a false satisfier;

*Professor, Computer Science and Automation, Indian Institute of Science, Bangalore, India 560 012; chandru@csa.iisc.ernet.in
†Professor, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213 USA; jh38@andrew.cmu.edu
‡Dip. di Informatica, University of Pisa, Corso, Italia 40, 56100 Pisa, Italy; rago@di.unipi.it
§Analyst, Sanchez Computer Associates (BDL), 37/2, 4th Main, Malleswaram, Bangalore, India 560 003; anjul@cyberspace.org

3. $P(t)$ and $P(t')$ have a most general unifier (σ, τ), such that $P(t)\sigma = P(t')\tau$.

4. There are clauses C, C' in F of which $P(t)$ and $P(t')$ are respectively satisfiers, and for which either $C\sigma$ or $C'\tau$ generalizes no clause in F.

5. The nesting depth of blockage is the level of the Herbrand universe at which the blockage is observed.

M-blockage: If $P(t)$ and $P(t')$ are blocked, and the nesting depth of the blockage is no greater than M, then we say that $P(t)$ and $P(t')$ are M-blocked.

M-satisfiability: If there are no predicates in F that are M-blocked in a particular satisfier mapping, then F is said to be M-satisfiable.

Let $F = \forall x_1 C_1 \wedge \ldots \wedge \forall x_m C_m$ be a first order formula.

1. Initialization. Set $F_0 := C_1 \wedge \ldots \wedge C_m$, $k := 0$, and $M := 0$.

2. Ground satisfiability. Try to find a satisfier mapping S for F_k that treats variants of the same atom as the same atom.

3. Termination check.

   (a) If S does not exist, then stop: F is unsatisfiable.

   (b) Otherwise, if S is unblocked, then stop: F is satisfiable.

   (c) Otherwise, if S is not M-blocked, then F is M-satisfiable. Let $M := M + 1$ and repeat Step 3(c).

4. Refinement. (S is M-blocked.) Let C_h and C_i be two clauses in F_k whose satisfiers are M-blocked, and let (σ, τ) be a most general unifier of $S(C_h)$ and $S(C_i)$. Set $F_{k+1} := F_k \wedge C_{h\sigma} \wedge C_{k\tau}$ after standardizing apart, set $k := k + 1$, and go to step 2.

It is possible for there to be more than one satisfier mapping S, yet if the particular S found in Step 2 is not M-blocked, we can claim that F is M-satisfiable. Although a different S could be M-blocked, that M-blockage would be resolved and ultimately F would be found to be M-satisfiable.

1.1 Unsatisfiability

Let $F = C_1 \wedge \forall x C_2 \wedge \forall y C_3$ where,

$$C_1 = P(s(a))$$
$$C_2 = \neg P(x) \vee Q(s(x))$$
$$C_3 = \neg P(y) \vee \neg Q(s(y))$$

The satisfier mapping is not 0-blocked, but the 1-blockage between $P(s(a))$ and $P(x)$ creates $F_1 = F_0 \wedge C_4$, with

$$C_4 = \neg P(s(a)) \vee Q(s(s(a)))$$

The satisfier mapping can be extended as shown. The 2-blockage between $Q(s(y))$ and $Q(s(s(a)))$ creates $F_2 = F_1 \wedge C_5$, with

$$C_5 = \neg P(s(a)) \vee \neg Q(s(s(a)))$$

Because F_2 is unsatisfiable as a propositional formula, the algorithm terminates.

2 Propositional Satisfiability Algorithms

Propositional satisfiability algorithms are very important in partial instantiation methods for first order theorem proving because in every iteration, we need to solve a propositional satisfiability problem obtained by treating variants of first order predicates as the same atom. Incremental propositional satisfiability algorithms play an even more important role, because in every iteration, the propositional satisfiability problem is the same as that in the previous iteration except for the addition of one or two clauses.

2.1 The DPL algorithm

We use the DPL algorithm [7] because of its generality, and excellent performance when coupled with the Jeroslow-Wang (JW) heuristic for choosing an atom from the formula. The JW heuristic tells us to pick the literal that occurs in the greatest number of short clauses. That is, if $\neg a$ occurs in 100 clauses of length 3, and does not occur in any shorter clauses, and $\neg b$ occurs in 1 clause of length 2, then the JW heuristic tells us to pick $\neg b$. That is, priority is given to the length of the clauses rather than to the number of occurrences. If the length of the shortest clauses is the same for two literals, then we consider the number of occurrences. Note that by "picking a literal", we mean choosing an atom and a "sign". Suppose we pick $\neg a$, then we generate two sub formulas by setting a to false and true. However, since we picked $\neg a$, we *first* recursively explore the formula generated by setting a to false. Had we chosen a instead of $\neg a$, we would first explore the formula generated by setting a to true.

2.2 Incremental DPL algorithm

It is often useful, especially with an implementation of PPI, to solve a propositional satisfiability problem incrementally. That is, having solved a satisfiability problem for a formula F, we want to solve it for a formula $F \wedge C$ where C is a new clause which may or may not contain an atom that does not exist in F. Note that if F is unsatisfiable, so is $F \wedge C$. Therefore, the non-trivial case is that F was found to be satisfiable, which means that the depth-first tree search was partially completed. In Incremental DPL, we avoid "re-searching" the part of the tree that was already searched while testing F for satisfiability. While solving F for satisfiability, we save the "state" of the search when we determine satisfiability, and continue from that point when we are testing $F \wedge C$. If C is true under the satisfying assignment found for F, then all is fine. If it is falsified, then we backtrack to the node in the tree where C is falsified (easily done with a little bookkeeping), and then continue the tree search.

If the truth value of C is undetermined, then we continue the search by choosing a variable from C and constructing sub-formulas as usual.

The incremental DPL algorithm gives excellent results for most problems in general, and causes a tremendous speed-up in the PPI algorithm.

3 A description of the algorithm and associated techniques

3.1 Introduction

There are two schools of practice in instantiation based methods, differing in the techniques used to control instance generation. Uncontrolled instance generation as used in early

attempts was not very successful because of combinatorial explosion of the number clauses.

One way of controlling instance generation is the use of semantics to guide the search of theorem provers. Early provers [17], [16], [3], allowed somewhat limited semantics. CLIN [11] uses a strategy based on clause linking. CLIN-S [3] added semantic support to clause linking. RRTP [12] generates instances using replacement rules and uses a propositional decision procedure [18] to detect propositional unsatisfiability. Ordered semantic hyper linking [13] improves upon CLIN and CLIN-S by allowing any semantics that can be expressed as a ground decision procedure, and imposes an ordering on all ground literals that allows OSHL to have a more systematic model generation strategy. It also permits the use of term rewriting to handle equalities.

The other school of thought is the Jeroslow school that uses blockage testing as a way of control over generation of instances. This method has recently been extended to full first order logic and a version is described in this section.

3.2 A Description of the Algorithm

The set of clauses that form a first order formula in Skolem normal form is a terse representation of a (usually infinite) set of ground clauses that are equivalent to a (usually infinite) formula in Propositional Logic. The compactness theorem [2] says that for an unsatisfiable first order formula, there is a finite set of ground clauses which is unsatisfiable as a propositional formula. This finite set is constructed by substituting variables with all constants whose depth is less than a particular finite value (initially unknown).

This leads us to the Instantiation procedure for testing satisfiability wherein we construct a set of ground clauses by substituting variables with constants upto depth 0, test for satisfiability, construct another set by substituting variables with constants upto depth 1, test, ..., construct another set by substituting variables with constants upto depth n and test again. If the formula is unsatisfiable, we are guaranteed to find that the set is unsatisfiable when we construct it by substituting variables with constants of some finite depth.

Unfortunately, the instantiation procedure is impractical because the number of clauses become astronomically large when we substitute variables with constants upto even a small depth (say 4-5).

While the compactness theorem says that there exists an unsatisfiable set of clauses generated by "grounding" upto a finite depth, it does not say that this set is minimal. In fact, this set is often not minimal, and the minimal set of ground clauses is often much smaller and is practical to handle. The PPI method tries to generate this minimal set and thereby determine unsatisfiability without attempting to handle the astronomical number of clauses.

We can see that atoms with universally quantified variables represent a (usually infinite) set of ground atoms.

At each stage the PPI algorithm solves a propositional satisfiability problem consisting of the universally qualitified formula without its quantifiers (but with Skolem functions generated by existential quantifiers in the original formula). Some of the atoms are only partially instantiated, but all atoms are treated equally as atomic propositions. Atoms that are *variants* of each other, however, are regarded as identical (two atoms are variants if they are the same but for renaming of variables). The aim is to assign truth values to some of the atoms of the formula so that at least one literal in every clause (the *satisfier* of the clause) is true. The complete instatiations of a satisfier are assumed to inherit the satisfier's

truth value. This provides a satisfying truth assignment for all complete instantiations of the formula, unless there is *blockage*; that is, unless satisfiers that are assigned different truth values have common instantiations, which inherit conflicting truth values. In such cases the clauses C_1, C_2 containing the conflicting satisfiers are further instantiated by the mgu of the satisfiers to generate clauses C_1', C_2', which are added to the formula. When the propositional satisfiability is re-solved, the conflict is resolved because the instantiations that once received conflicting truth values now ingerit their truth value from the atoms in C_1', C_2'.

To ensure finite termination of the PPI algorithm for unsatisfiable formulas, we need to modify the procedure by resolving, among the several blockages that may exist at the same point of time, any one of those having the least nesting depth of the mgu. By doing this, we ensure that we fully explore the set of ground clauses obtained using constants only upto a certain depth, before proceeding to the next level.

It is important to see here the role of propositional satisfiability algorithms in PPI, particularly that of the Incremental DPL algorithm. The assignment of non-conflicting truth values to an atom in each clauses is done by solving a propositional satisfiability problem wherein the FOF is treated as a propositional formula with variants of atoms corresponding to the same literal in the propositional formula. Since we solve such problems many times adding one or two clauses each time, it is imperative that we use an incremental propositional satisfiability algorithm. An incremental algorithm reuses the earlier computation and thereby *usually*saves a large amount of time. In our program we implemented an incremental version of the DPL algorithm as described by [7].

3.3 Incremental Blockage Testing

We see that the PPI procedure involves repeatedly testing for the existence of blockages after solving propositional satisfiability problems. While the time consumed in repeatedly solving propositional satisfiability problem solving was drastically reduced by using the Incremental DPL algorithm, there was still the bottleneck of testing for blockage.

Blockage testing is done by checking whether the satisfiers of any pair of clauses are unifiable, and if so, whether the substituted clauses are generalized by any clause in the formula. The unification test involves $O(n^2)$ attempts at unification (for all pairs of clauses), whereas checking for the existence of a generalizing clause needs $O(n)$ generalization tests whenever unification is successful. Here n is the number of clauses in the current formula.

We observed that quite often, the satisfier atoms for most clauses are the same as in the previous iteration (i.e. the last time the propositional satisfiability problem was solved). This led to the idea of testing blockages also incrementally by re-using the results of the blockage tests (mgus and generalizations) that were obtained in the previous iteration.

This incremental testing of blockages speeded up the entire procedure considerably by saving unification attempts, and generalization tests in many cases.

To implement incremental blockage testing, we associate with each clause, a list of results with a node for each subsequent clause. Each result node stores a pointer to the satisfiers of both clauses when the tests were last carried out, and the results of the tests (i.e. whether the mgu was found, if so, what was the mgu, and whether the generalization test succeeded or not). The very first time, all these nodes are empty, but as we perform blockage tests, we store the results in this list. On subsequent occassions of blockage testing between a pair of clauses, we look up the appropriate result node and check whether the satisfiers are still the same. If yes, then we simply lookup the mgu and generalization test

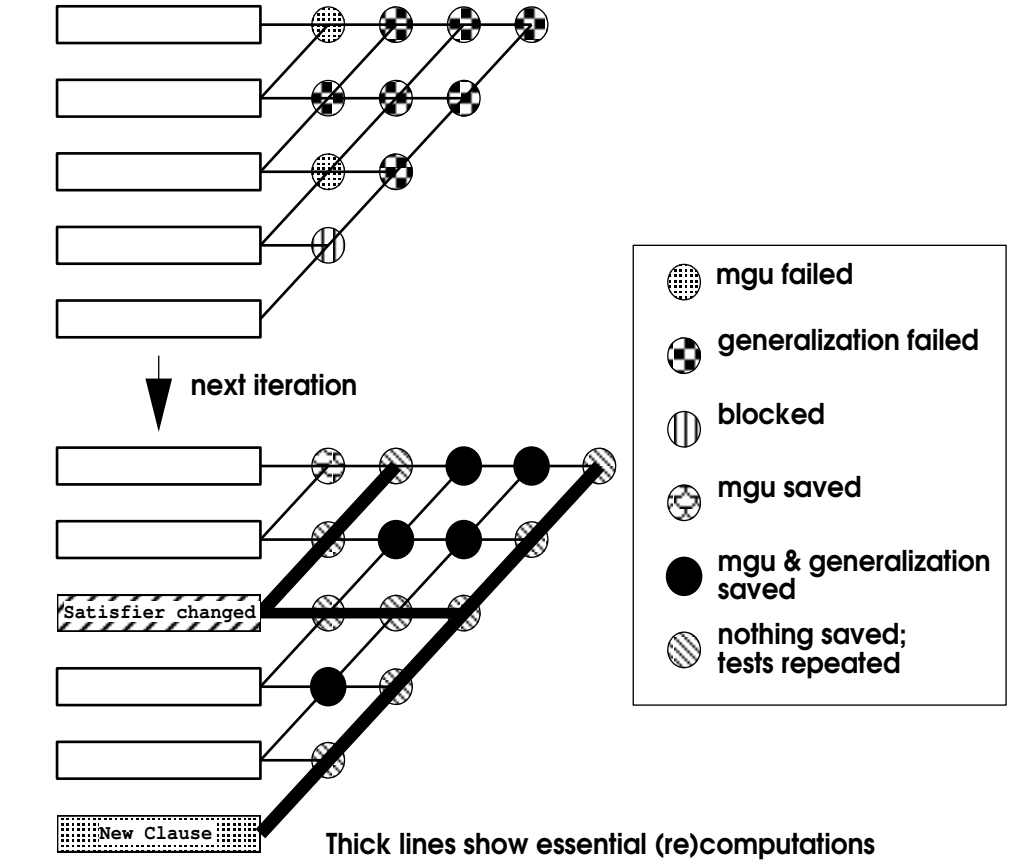

Figure 1: Incremental Blockage Testing

results. If the satisfiers have changed, we cannot use the results and have to re-do the computations. However, it very often happens that the satisfiers are the same as the last time and therefore we can reuse the results.

Note that whenever a clause is added, we have to add an empty result node to each of the clauses in the formula. Also, the generalization test is technically not entirely saved because a check is needed whether the newly added clause generalizes the substituted clauses. However, for all practical purposes, we can assume that the generalization test is saved because we have just one test where we would have needed n tests.

4 Results

4.1 Performance improvement due to Incremental Techniques

Problem Class		A	B	C	D	E
Monkey-Banana	:	5	25	0.24	0.30	0.32

```
Shortest Plan Example :  13  38    0.27    0.43    0.70
Blocks World (Smaller):  19 118    2.53    5.48   27.95
Blocks World (Larger) :  33 279   16.31   47.05 1904.02
=======================================================
A: Number of Initial Clauses
B: Number of Final Clauses (# Blockages <= B-A)
C: CPU seconds for Incremental SAT and Blockage
D: CPU seconds for only Incremental SAT
E: CPU seconds for no incremental components
=======================================================
```

4.2 Planning Problems: The Monkey Banana Problem

The following pages show summarized and detailed solutions of the monkey-banana problem which is a standard and representative planning problem commonly used to study methods of solution of planning problems. These solutions were obtained by using PPI to implement the first-order theorem proving technique for planning.

The method resolved 20 blockages, making the final number of clauses 25. There were 15 literals in the final propositional satisfiability problem.

Therefore the method needed to solve a 25 clause, 15 literal propositional satisfiability problem incrementally (in 20 steps). Also, at each step it had to test for blocked predicates. This test needs, in the worst case, an $O(n^2)$ mgu computations, and $O(n^3)$ generalization sub-tests (assuming $O(n^2)$ mgu computations succeed), where n is the number of clauses in the formula at that point. Since this test is done m times, we have $O(m^3)$ mgu computations, and $O(m^4)$ generalization tests. The generalization tests are performed $O(n)$ times only if an mgu computation is successful, therefore, the $O(m^4)$ complexity should be treated with great caution.

Profiling results showed 427 mgu computations (both successful and unsuccessful), and 4027 generalization tests (when the mgu test was successful).

It is difficult to perform a complexity analysis because we cannot estimate how many blockages will need to be resolved before a particular problem is solved. It may be that we can come up with an unblocked satisfier assignment right in the beginning, or have to resolve several blockages before this happens.

4.2.1 Summarized solution

The problem is stated below as a first order formula. The first four clauses are the rules, and the last clause is the negation of the query. The PPI method generates the answer by determining the formula to be unsatisfiable.

```
-P:1(x,y,z,s)+P:2(z,y,z,k(x,z,s))
-P:1(x,y,x,s)+P:3(y,y,y,y(x,y,s))
-P:4(B,B,B,s)+R:5(b(s))
+P:6(A,B,C,S)
-R:7(s)
```

$P(x, y, z, s)$ means that the monkey is at position x, ladder is at position y, and the banana is at position z, in state s.

$R(s)$ means that the monkey can grasp the banana in state s.

k, y, b are abbreviated names for the functions: $walk, carry, climb$ respectively.

The number to the right of a predicate name is the propositional atom identification to which it corresponds to. I.e. it is an identification for the equivalence class of variants to which the predicate belongs.

The following are the additional clauses added upon resolution of various blockages encountered. After adding 20 clauses, the formula is determined to be unsatisfiable.

```
-P:6(A,B,C,S)+P:8(C,B,C,k(A,C,S))
-P:8(C,B,C,k(A,C,S))+P:9(C,B,C,k(C,C,k(A,C,S)))
-P:8(C,B,C,k(A,C,S))+P:10(B,B,B,y(C,B,k(A,C,S)))
-P:9(C,B,C,k(C,C,k(A,C,S)))+P:11(C,B,C,k(C,C,k(C,C,k(A,C,S))))
-P:10(B,B,B,y(C,B,k(A,C,S)))+P:12(B,B,B,k(B,B,y(C,B,k(A,C,S))))
-P:9(C,B,C,k(C,C,k(A,C,S)))+P:13(B,B,B,y(C,B,k(C,C,k(A,C,S))))
-P:10(B,B,B,y(C,B,k(A,C,S)))+P:14(B,B,B,y(B,B,y(C,B,k(A,C,S))))
-P:10(B,B,B,y(C,B,k(A,C,S)))+R:15(b(y(C,B,k(A,C,S))))
-P:11(C,B,C,k(C,C,k(C,C,k(A,C,S))))+P:16(C,B,C,k(C,C,k(C,C,k(C,C,k(A,C,S)))))
-P:12(B,B,B,k(B,B,y(C,B,k(A,C,S))))+P:17(B,B,B,k(B,B,k(B,B,y(C,B,k(A,C,S)))))
-P:13(B,B,B,y(C,B,k(C,C,k(A,C,S))))+P:18(B,B,B,k(B,B,y(C,B,k(C,C,k(A,C,S)))))
-P:14(B,B,B,y(B,B,y(C,B,k(A,C,S))))+P:19(B,B,B,k(B,B,y(B,B,y(C,B,k(A,C,S)))))
-P:11(C,B,C,k(C,C,k(C,C,k(A,C,S))))+P:20(B,B,B,y(C,B,k(C,C,k(C,C,k(A,C,S)))))
-P:12(B,B,B,k(B,B,y(C,B,k(A,C,S))))+P:21(B,B,B,y(B,B,k(B,B,y(C,B,k(A,C,S)))))
-P:13(B,B,B,y(C,B,k(C,C,k(A,C,S))))+P:22(B,B,B,y(B,B,y(C,B,k(C,C,k(A,C,S)))))
-P:14(B,B,B,y(B,B,y(C,B,k(A,C,S))))+P:23(B,B,B,y(B,B,y(B,B,y(C,B,k(A,C,S)))))
-P:12(B,B,B,k(B,B,y(C,B,k(A,C,S))))+R:24(b(k(B,B,y(C,B,k(A,C,S)))))
-P:13(B,B,B,y(C,B,k(C,C,k(A,C,S))))+R:25(b(y(C,B,k(C,C,k(A,C,S)))))
-P:14(B,B,B,y(B,B,y(C,B,k(A,C,S))))+R:26(b(y(B,B,y(C,B,k(A,C,S)))))
-R:15(b(y(C,B,k(A,C,S))))

UNSATISFIABLE.

Answer: The monkey has the banana after he walks from A to C, carries
the ladder from C to B,
and then climbs the ladder.
```

4.3 Planning Problems: Shortest Plan Generation Property

We use limited answer generation techniques to solve planning problems. To do this, we add an answer predicate, ANS, which is treated differently from normal predicates by the PPI implementation. We use the ANS predicate in the query clause to specify the variable that is generating the answer.

The ANS predicate is ignored during the satisfier assignments. That is, it can never be a satisfier for a clause. It is also ignored during the generalization tests. That is, we delete the ANS predicate if it exists in a clause before testing a pair of clauses for generalization. For all other operations, especially for substitution and addition of substituted clauses to remove blockage, we treat the ANS predicate normally.

This scheme of answer generation works for a limited subset of first order formulas. It works for Horn formulae, and therefore we use it for answer (plan) generation in planning problems.

To ensure termination in a finite number of steps, the PPI explores the Herbrand Universe level by level (breadth-first). This feature causes it to generate the shortest plans when it is used to solve planning problems.

4.4 An Example of Shortest Plan Generation

You can move between positions A through G in single steps or jumps of two. How do you go from A to F in the shortest number of steps?

```
+P:1(A,S)                  #Initially at A
-P:2(A,s)+P:3(B,step(s))   #Rule: Can step from A to B
-P:4(B,s)+P:5(C,step(s))   #Rule: Can step from B to C
-P:6(C,s)+P:7(D,step(s))   #Rule: Can step from C to D
-P:8(D,s)+P:9(E,step(s))   #Rule: Can step from D to E
-P:10(E,s)+P:11(F,step(s)) #Rule: Can step from E to F
-P:12(F,s)+P:13(G,step(s)) #Rule: Can step from F to G
-P:2(A,s)+P:14(C,jump(s))  #Rule: Can step from A to C
-P:4(B,s)+P:15(D,jump(s))  #Rule: Can step from B to D
-P:6(C,s)+P:16(E,jump(s))  #Rule: Can step from C to E
-P:8(D,s)+P:17(F,jump(s))  #Rule: Can step from D to F
-P:10(E,s)+P:18(G,jump(s)) #Rule: Can step from E to G
-P:12(F,s)+ANS:0(s)        #Query: Cannot move from A to F

... after several blockage resolutions ...

UNSATISFIABLE
```

The answer generated: $step(jump(jump(S)))$ is a shortest way to go from A to F. A longer way is: $step(step(step(jump(S))))$.

5 Conclusions

Blockage testing is the current bottleneck. As blockages are removed, clauses are added to the formula and the time taken to test for blockage increases. The time complexity (of blockage testing) is between $O(n^2)$ and $O(n^3)$; the former holds in the case where all the mgu tests fail, and the latter where they all succeed. Therefore, it is difficult to solve complex problems because of this slowdown.

Another problem is that the number of blockages resolved before reaching unsatisfiability depends upon the order in which they are removed, and upon the way the satisfiers are assigned. Therefore, there is reason to search for heuristics that help choose such satisfier mappings, and cause such blockages to be resolved, that unsatisfiability is reached with a near minimum number of blockages resolved.

References

[1] Anjul Shrivastava, *A study of the Partial Instiation Method*, Masters thesis, Department of Computer Science, IISc, Aug 1998

[2] Chang, C. and Lee, R. C. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[3] Chu, H., and Plaisted, D., *Semantically guided first-order theorem proving using hyper-linking* Proceedings of the Twelfth International Conference on Automated Deduction, 192-206. Lecture Notes in Artificial Intelligence 814.

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

[5] Gallo, G. and G. Rago, *A hypergraph approach to logical inference for datalog inference*, Tech. Rep. 28/90, Dip. Informatica, Università di Pisa (1990).

[6] Gallo, G. and G. Rago, *The satisfiability problem for the Schöenfinkel-Bernays fragment: partial instantiation and hypergraph algorithms*, Tech. Rep. 4/94, Dip. Informatica, Università di Pisa (1994).

[7] Hooker, J. *Solving the Incremental Satisfiability Problem.* Journal of Logic Programming 15(1993)

[8] Hooker, J., *New methods for computing inferences in first order logic*, Annals of Operations Research 43 (1993) 479-492.

[9] Hooker, J., Rago, Gabriela *Partial Instantiation Methods for Inference in First Order Logic.* Working Paper, Graduate School of Industrial Administration, CMU, Dec 1996

[10] Jeroslow, R. G., *Computation-oriented reductions of predicate to propositional logic*, Decision Support Systems 4 (1988) 183-197.

[11] Lee, S. J., and Plaisted, D., *Eliminating duplication with the hyper-linking strategy* Journal of Automated Reasoning 1:103-114.

[12] Paramasivam, M., and Plaisted, D., *A Replacement Rule Theorem Prover* Journal of Automated Reasoning Forthcoming.

[13] Plaisted, D., and Zhu, Y., *Ordered Semantic Hyper Linking* Proceedings of Fourteenth National Conference on Artificial Intelligence (AAAI-97).

[14] Plaisted D. and Yunshan Zhu *FOLPLAN: A Semantically Guided First-Order Planner*, 10th International FLAIRS Conference, Daytona Beach, Florida, May 11-14, 1997

[15] Robinson, J. A. *Logic and Logic Programming.* Communications of the ACM, July 1992.

[16] Slagle, J. R., *Automatic theorem proving with renamable and semantic resolution* Journal of the ACM 14(4):687-697.

[17] Wang, T., and Bledsoe, W., *Hierarchical deduction*, Journal of Automated Reasoning 3:35-77.

[18] Zhang, H., and Stickel, M. E., *Implementing the Davis- Putnam algorithm by tries* Technical report, Department of Computer Science, University of Iowa