ABSTRACT

Title of Thesis: Performance of BLACKBOX Planning System on a Hard Problem of Satisfiability

Name of degree candidate: Kumar Shashi Prabh

Degree and date: Master of Science, May 2001

Thesis directed by: Professor Ernest Davis
Department of Computer Science

BLACKBOX is one of the best SAT-based planning system available at present. The system's performance on a supposedly hard problem is presented. The problem is to find values of unbound variables that make a given Problem Goal true. The variables take two values. The goal of the problem is in the form of 3-CNF sentences, e.g., $(A \vee C \vee F) \wedge (\neg A \vee B \vee \neg L) \wedge \ldots$. Algorithms are described to find a model for the goal, and to encode the problem in a format that is compatible with the input restrictions of BLACKBOX.

Performance of BLACKBOX Planning System on a Hard Problem of Satisfiability

by

Kumar Shashi Prabh

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

New York University

May 2001

Advisory Committee:

Professor Ernest Davis, Advisor

Professor Davi Geiger

© Copyright by

Kumar Shashi Prabh

May 2001

TABLE OF CONTENTS

List of Tables iv

List of Figures vi

1 Introduction 1

1.1 Planners based on First-Order Logic 1

1.2 Propositional Planners 2

2 Propositional Planning: The State of the Art 4

2.1 Graphplan 4

2.1.1 Planning Graph 4

2.1.2 The Graphplan Algorithm 6

2.2 Planning as Satisfiability 8

2.3 BLACKBOX 10

3 Experiment 13

3.1 The Problem 13

3.2 Algorithm 14

3.3 Including Typing 20

3.4 Carrying On 26

4 Conclusion and Further Study **33**

5 Appendix **35**

5.1 Code for Generating Untyped Problem Instances 35

5.2 Code for Generating Typed Problem Instances 41

5.3 Code for Generating SAT-Like Problem Instances 48

Bibliography **56**

LIST OF TABLES

3.1 Time taken by BLACKBOX to find a plan for problems containing 20 variables 16

3.2 Time taken by BLACKBOX to find a plan for problems containing 30 variables 17

3.3 Time taken by BLACKBOX to find a plan for problems containing 40 variables 17

3.4 Time taken by BLACKBOX to find a plan for problems containing 50 variables 17

3.5 Time taken by BLACKBOX to find a plan for problems containing 20 variables and typing 22

3.6 Time taken by BLACKBOX to find a plan for problems containing 40 variables and typing 22

3.7 Time taken by BLACKBOX to find a plan for problems containing 60 variables and typing 23

3.8 Time taken by BLACKBOX to find a plan for problems containing 80 variables and typing 23

3.9 Time taken by BLACKBOX to find a plan for problems containing 100 variables and typing 24

3.10 Time taken by BLACKBOX to find a plan for problems containing 20 variables and typing, variables initialized to true 27
3.11 Time taken by BLACKBOX to find a plan for problems containing 40 variables and typing, variables initialized to true 27
3.12 Time taken by BLACKBOX to find a plan for problems containing 100 variables and typing, variables initialized to true 28
3.13 Time taken by BLACKBOX to find a plan for problems containing 200 variables and typing, variables initialized to true 28
3.14 Time taken by BLACKBOX to find a plan for problems containing 300 variables and typing, variables initialized to true 29

LIST OF FIGURES

2.1 Example of a Planning Graph 6

2.2 SAT Planning System Architecture 9

3.1 Plot of the time taken (on logarithmic scale) to find a plan for problems containing 20 variables vs. the ratio of the number of sentences and the number of variables 18

3.2 Plot of the time taken by WALKSAT to find a plan vs. the number of sentences. The number of variables is the parameter 18

3.3 Plot of the time taken by randomized SATZ (the data for SATZ is similar) to find a plan vs. the number of sentences. The number of variables is the parameter 19

3.4 Plot of the time taken by Graphplan to find a plan vs. the number of sentences. The number of variables is the parameter 19

3.5 Plot of the time taken (on logarithmic scale) to find a plan for typed problems containing 20 variables vs. the ratio of the number of sentences and the number of variables 24

3.6 Plot of the time taken by WALKSAT (the data for SATZ, and RAND-SATZ is similar) to find a plan vs. the number of sentences. The number of variables is the parameter 25

3.7 Plot of the time taken by Graphplan to find a plan for typed problems vs. the number of sentences. The number of variables is the parameter 25

3.8 Plot of the time taken to find a plan for typed problems containing 20 variables vs. the ratio of the number of sentences and the number of variables (initialized to true) 29

3.9 Plot of the time taken by WALKSAT to find a plan vs. the number of sentences. The number of variables (initialized to true) is the parameter 30

3.10 Plot of the time taken by SATZ to find a plan vs. the number of sentences. The number of variables (initialized to true) is the parameter 30

3.11 Plot of the time taken by RAND-SATZ to find a plan vs. the number of sentences. The number of variables (initialized to true) is the parameter 31

Chapter 1

Introduction

Planning addresses the problem of automatically generating a sequence of possible actions, called a plan, that achieves specified set of goals beginning with a given set of initial states. A reliable and fast planning capability is often critical to the effectiveness of an intelligent agent.

1.1 Planners based on First-Order Logic

One of the earliest methods used to solve this problem was Situation Calculus, developed by McCarthy and Hayes [10]. It is a predicate calculus formalization of states, actions and their effects [12, page 363]. The major difficulty of this approach was the so called *frame problem*. A frame axiom tells which of the fluents don't change as the result of an action. For example, if there are three blocks A, B, and C stacked on top of each other, C being at the topmost position, then moving C from B to the table has no effect on A.

> Frame axioms are used to prove that a property of a state remains true if the state is changed by an action that does not affect that property . . . Since, typically, we will have a pair of frame axioms for

every combination of fluent and action, it becomes unmanageably difficult in realistic problems to represent how actions change the world in this formulation of the situation calculus [12, page 368].

The problem of large number of frame axioms, and to automatically generate them in the Situation Calculus approach led to an alternative approach proposed by Fikes and Nilsson [4] called STRIPS, the Stanford Research Institute Problem Solver. It treats the *predicate formulas* describing the state as "a kind of state." The search is performed in "Plan Space" rather than in "State Space" (as opposed to the Situation Calculus approach). Actions are represented as *operators* which consist of three parts: preconditions, add list, and delete list. An action is exectuted only when all the literals in the preconditions set are present in the state description. The effect of an action is to delete all the literals of the delete list from the state description, and to add all the literals of the add list to it. A plan corresponds to sequence of operators that achieve the goal state given the starting state.

In these schemes, finding the plan is basically *searching* for a sequence that will lead to the goal from the initial state. In practice certain action(s) will appear more than once in the plan. In the simplest case, assuming that all the actions appear not more than once in the plan, the number of possible seuences is factorial of the number of actions. This means that search in this scenario is an NP-Hard problem.

1.2 Propositional Planners

Another approach to planning is "Satisfiability" (SAT) [6]. The main idea behind propositional planning scheme is to first compile the initial state(s), the goal(s), and the action domain into a set of propositional formulae, and then find a satisfiability

(truth) assignment using some fast satisfiability algorithm (e.g. GSAT), and then extract the plan back from the assignments. Satisfiability is believed to be a much harder problem than deduction - the approach that was used in situation calculus and STRIPS . However, using fast SAT algorithms like GSAT, for searching more than makes up for the deficiency. This concept will be further elaborated in the next chapter.

Chapter 2

Propositional Planning: The State of the Art

Research activity in the area of planning has risen greatly in the last few years, mainly because of new fast algorithms, efficient techniques, and better understanding [14]. Planning techniques have been (or are being) applied to robot movement, spacecraft control, scheduling operations in automotive industry, oil-spill response planning (by US Coast Guard), military air-campaign planning, human computer interaction, and many other areas [15].

2.1 Graphplan

The Graphplan planning system is based on the Planning Graph Analysis approach, developed by Blum and Furst [2]. When introduced, it achieved a significant breakthrough in trems of planning speed. It is regarded as one of the most successful encodings of planning problems as a SAT problem.

2.1.1 Planning Graph

According to [2], a *valid plan* for a planning problem consists of a set of actions, and specified times in which each action is to be carried out. More than one action is

allowed provided they don't interfere (i.e. one action does not delete a precondition or an add-effect of another) with each other. A valid plan may perform an action at time t if the plan makes all of its preconditions true at time t (at $t = 1$, it translates to the given initial conditions). There are "no-op" actions that carry truth forward in time without introducing any new effect. We may define a proposition to be true at time $t > 1$ iff it is an add effect of some action taken at time $t - 1$. And of course, a valid plan must make all the Problem Goals true at the end.

A Planning Graph (Figure 2.1) is similar to a valid plan except that it is *not* required to obey non-interference. A Planning Graph is a directed, and leveled graph (i.e., nodes can be partitioned into disjoint sets such that the edges connect nodes in adjacent levels only) with two kinds of nodes and three kinds of edges. The levels alternate between proposition levels containing proposition nodes, and action levels containing action nodes. Each of the proposition and action nodes are labeled with some propostion and some action respectively. The first level of a Planning Graph is a proposition level and consists of one node for each proposition that is given in the initial conditions. The levels are arranged in a sort of pairwise fashion: true propositions at time 1, possible actions at time 1; true propositions at time 2, possible actions at time 2 etc. Edges in a Planning Graph represent relations between actions and propositions through connecting action nodes to their preconditions in the previous proposition level by the so-called "Proposition-Edges", and to their effects in the following proposition level by "Add-Edges", and "Delete-Edges". Action level i may contain all the possible actions whose preconditions all exist in proposition level i. A proposition may exist at proposition level $i+1$ if it is an add-effect of some action of level i, even if this proposition may be a delete-effect of some other action of level i. Also, because of "no-op" actions, every propostion that appears at level i

may also appear at level $i + 1$. Since the requirements on Planning Graph are weak, they can be built very fast.

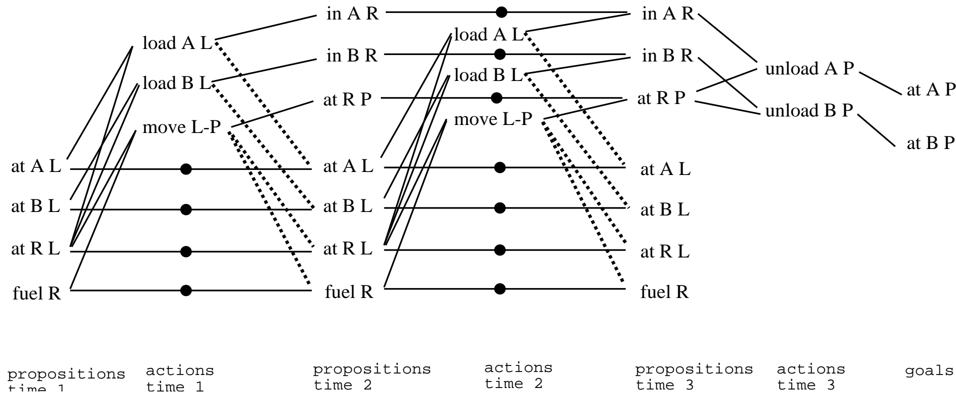

Figure 2.1: Example of a Planning Graph for the rocket domain. Source: [2].

2.1.2 The Graphplan Algorithm

Two actions are *mutually exclusive* at a given action level if no valid plan could possibly contain both. Similarly, two propositions at a given proposition level are mutually exclusive if no valid plan could possibly make both true. Graphplan marks two actions at a given action level to be exclusice of each other as follows:

Interference If either of the actions deletes a precondition or add-effect of the other.

Competing Needs If the precondition of action a and that of action b are marked mutually exclusive in the previous preposition level.

Two propositions p and q in a proposition level are marked as exclusive if all ways of creating proposition p are exclusive of all ways of creating proposition q.

The first proposition level of the Planning Graph consists of all the initial conditions. Then the Planning Graph is extended as follows. To create a generic

action level, for each operator and each way of instantiating preconditions of that operator to propositions in the previous level, an action node is inserted if no two of its preconditions are labeled as mutually exclusive. All the "no-op" actions and the precondition edges are inserted as well. Then the action nodes are checked for exclusivity. To create a generic proposition level, all the add-effects of the actions in the previous level including "no-ops" are placeed in the next level as propositions, connecting them via the appropriate add and delete edges. Any two propositions are marked as exclusive if all ways of generating the first are exclusive of all ways of generating the second.

The time taken by this algorithm to create this graph structure is polynomial in the length of the problem's description and the number of time steps. The part of graph creation that takes the most time is determining exclusion relations. An improvement to the basic algorithm described above is to avoid searching until a proposition level has been created in which all the final goals appear and no pair of goals has been determined to be mutually exclusive.

Given a Planning Graph, Graphplan searches for a valid plan using backward chaining strategy. Given a set of goals at time t, it attempts to find a set of actions ("no-ops" included) at time $t-1$ having these goals as add effects, i.e. the preconditions of the actions at a level becomes the goals of the previous level. Clearly, this set of subgoals at time $t-1$ has the property that if these goals can be achieved in $t-1$ steps, then the original goals can be achieved in t steps. If the goals set turns out to be unachievable, Graphplan tries to find a different set of actions. This process is continued until it either succeeds or it has proven that the original set of goals is not solvable at time t. As the next step, it either takes another Extend and Search step or terminates the search. Thus, in each iteration through this Extend and

Search loop, the algorithm either discovers a plan or else proves that no plan having that many time steps or fewer is possible. This algorithm is sound and complete in the sense that any plan the algorithm finds is a legal plan, and if there exists a legal plan then Graphplan will find one. It is also possible to augment this algorithm so that if the Problem Goals are not satisfiable by any valid plan, then the planner is guaranteed to halt with failure in finite time.

An important aspect of Graphplan's search is that when a set of (sub)goals at some time t is determined to be not solvable, then before popping back in the recursion it memoizes what it has learned, storing the goal set and the time t in a hash table. Similarly, when it creates a set of subgoals at some time t, before searching it first probes the hash table to see if the set has already been proved unsolvable. If so, it then backs up right away without searching further. This memoizing step, in addition to its use in speeding up search, provides necessary and sufficient conditions for terminating the search. Graphplan is being used as a preprocessor in some of the fastest planning systems based on compilation approaches – our next topic.

2.2 Planning as Satisfiability

There are three popular categories of planning systems that use "compilation of planning problems" approach, namely those that compile the problem to SAT (Satisfiability) encodings, CSP (Constraint Satisfaction Problem) encodings, and ILP (Integer Linear Planning) encodings. The basic idea behind the first two is as follows. The problem is first converted to a Planning Graph using Graphplan . The planning graph is then converted into a SAT wff (well formed formula), or CSP encoding, which then is solved using a SAT or CSP algorithm. In the third case, the SAT encoding is

translated to ILP inequalities, and then solved using some appropriate algorithm (for example, branch and bound algorithm, [8]. BLACKBOX [7] is an implementation of the SAT planning category. GP-CSP [3] and ILP-PLAN [8] are implementations of CSP and ILP categories respectively.

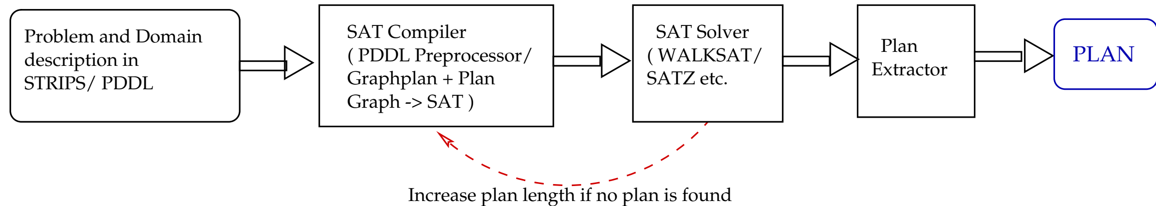

Figure 2.2: Architecture of a typical SAT-based planning system

In the following sections we shall consider Planning as SAT. Planning as SAT was developed in 1992 by Kautz and Selman [6]. According to them, the satisfiability approach provides a more flexible framework for stating different kinds of constraints on plans and also reflects the theory behind modern constraint based planning systems more accurately. As stated in the introduction, satisfiability is the problem of finding a model (truth assignments) of a set of axioms. In the planning as satisfiability approach, a planning problem is simply a set of axioms with the property that any model of the axioms corresponds to a valid plan. This approach was inspired by remarkable success of GSAT [13], a randomized greedy algorithm to solve satisfiability problems. In the formalization of planning as satisfiability, it is easy to state arbitrary facts about any state of the world, not just the initial and goal states. It is likewise easy to state arbitrary constraints on the plan – for example, that it contain a specified action performed at a specified time.

The formulation of Planning as Satisfiability is as follows. Every action and proposition has a time attached to them. For example, in the "Blocks World", the action move(A, B, C) means move block A from block B to block C. If this actions

takes place at time t, then it will be represented as move(A, B, C, t). This leads to an interesting problem of representing the problem goal. Let us assume that we want the block B at the end to be on the block A. Clearly, on(B, A) will not do the job. This leads to the restriction that we can encode the problem only for a *fixed* plan length. To rule out the possibility that an action executes despite the fact that its preconditions are false, it is asserted that an action implies its preconditions as well as its effects, e.g., $\forall_{a,b,c,t}\ move(a,b,c,t) \Rightarrow (clear(a,t) \wedge clear(c,t) \wedge on(a,b,t))$. It should be noted that in this formulation preconditions and effects are treated symmetrically. Secondly, only one action occurs at a time. $\forall_{a,a',b,b',c,c',t}\ (a \neq a', b \neq b', c \neq c') \Rightarrow \neg move(a,b,c,t) \vee \neg move(a',b',c',t)$. And finally, some action occurs at every time. This is not a significant restriction, since we can always introduce a "do nothing" action. In the Blocks World, this translates to $\forall_{t<N} \exists_{a,b,c}\ move(a,b,c,t)$.

One advantage of this approach is that it is easy to specify conditions in any intermediate state of the world, not just the initial and goal states. For example, if it is desired that block A be on either block B or D at time 25, one needs to simply add the assertion $on(A,B,25) \vee on(A,D,25)$ to the problem specification. Also, in this framework the plan requirements are all constraints on the models.

2.3 BLACKBOX

The BLACKBOX planning system [7] unifies the planning as satisfiability framework with the Graphplan planning approach to STRIPS planning. Becuase of the success of the planning as propositional satisfiability approach the belief that planning required specialized algorithms was challenged. SATPLAN, a planning system based on the satisfiability approach, showed that a general propositional theorem prover could

indeed be competitive with the specialized planning systems. There is considerable effort being made in the direction of creating faster SAT engines. Furthermore, planning community has settled on common representations that allow sharing and improvement of the algorithms and code. As result, the best general SAT engines are generally faster than the best specialized planning engines.

Graphplan shares a number of features with with the SATPLAN. Comparisons to SATPLAN show that neither approach is strictly superior. For example, SATPLAN is faster on a complex logistics domain, they are comparable on the blocks world, and on several other domains Graphplan is faster [14]. SATPLAN takes as input a set of axiom schemas where as Graphplan takes a set of STRIPS -style operators. Despite this practical difference, they have inmportant similarities. Both first create a propositional structure - Graphplan creates a Plan Graph, and SATPLAN creates a CNF wff - and then they perform a search over assignments to variables that is constrained by the structure they have created. As mentioned before, the propositional structure corresponds to a fixed plan length, and the search reveals whether a plan of that length exists. The authors of BLACKBOX had found that the Plan Graph had a direct translation to CNF, and that the form of the resulting formula was very close to the original conventions for SATPLAN. According to [5], SATPLAN and Graphplan , alongwith CSP and ILP based planning systems are examples of disjunctive planners. The initial creation of the propositional structure is a case of plan refinement without splitting, while the search through the structure is a case of plan extraction. [7] hypothesize that the differences in performance of the two sys tem can be explained by the fact that Graphplan uses a better algorithm for instantiating (refining) the propositional structure, while SATPLAN uses more powerful search (extraction) algorithms. BLACKBOX combines the best features of

Graphplan and SATPLAN. It works in a series of phases as follows. First, a planning problem that is specified in the Planning Domain Definition Language (PDDL) [11] is fed to a preprocessor that translates the problem into STRIPS style ready for inputting to a modified version of Graphplan , that acts as the front end of the system. The problem encoding may not contain any disjunctions in goals, conditional goals, universal quantifiers or existential quantifiers, even though they are valid constructs in PDDL. Preprocessed output is then converted to a plan graph of length k, and mutexes are computed. Then Plan Graph is converted to a CNF wff. The resulting CNF wff is simplified by a general CNF simplification algorithm. The simplified wff is then solved by any of a variety of fast SAT engines like WALKSAT, SATZ [9], and RELSAT [1]. If a model of the wff is found, then the model is converted to the corresponding plan else k is incremented and the process repeats. It is also possible to let Graphplan alone find the plan.

Chapter 3

Experiment

3.1 The Problem

There are some given unbound variables initially, and our aim is to find values of the variables that makes the Problem Goal true. The values that the variables can take is restricted to either true or false, or in other words, the variables take two values. The goal of the problem is in the form of 3-CNF sentences, e.g., $(A \vee C \vee F) \wedge (\neg A \vee B \vee \neg L) \wedge \ldots$. We want to find a model for the goal, using BLACKBOX. The problem can be viewed as a problem of satifiability posed as a planning problem ("Satisfiability as Planning" !). In Planning Domain Definition Language (PDDL), the previous example can be written as:

```
(:goal (and  (or  (value A true)
(value C true)
(value F false))
(or  (value A false)
(value B true)
(value L false))))
```

BLACKBOX does not support "or" (disjunctions). Therefore, we need to re-encode the goal so that it can be fed to BLACKBOX. The following section describes the algorithms.

3.2 Algorithm

My approach to this problem was to initialize all the unbound variables with a value called "unbound", and then define an action that assigns these variables either truth or falsehood. The following code illustrates this:

```
(:init (value A unbound)
(value B unbound) ... )


;;; assign true or false to the variables
(:action SET-VAR
:parameters  (?var ?value)
:precondition  (value ?var unbound)
:effect  (and (value ?var ?value)
(not (value ?var unbound))))
```

Next, to encode disjunctions in a form that can be processed by BLACKBOX, for each of the disjunctions we generate actions for each of the clauses. The action has the clause as the precondition, and the effect is assigning truth value to satisfaction of the part of the goal that corresponds to the sentence within the scope of the disjunction. To illustrate:

```
 (:goal (and  (or  (value A true)
```

```
(value C true)
(value F false))
(or  (value A false)
(value B true)
(value L false))))
```

is translated to

```
(init:  (value goal1 false)
(value goal2 false) ... )

(:action OR1
:precondition (value A true)
:effect  ( and (value goal1 true)
(not (value goal1 false))))

(:action OR12
:precondition (value C true)
:effect  ( and (value goal1 true)
(not (value goal1 false))))
...

(:action OR22
:precondition (value B true)
:effect  ( and (value goal2 true)
(not (value goal2 false))))
...
```

Sentence Size	Varsize: 20			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.47	0.46	0.46	0.50
40	1.00	0.98	1.00	1.15
60	2.21	1.82	2.12	1.92
80	2.94	2.98	2.97	4.92
100	30.52	4.85	4.55	–

Table 3.1: Time taken by BLACKBOX to find a plan for problems containing 20 variables

```
    (:goal (and (value goal1 true)
(value goal2 true) ... ))
```

The experiments were run on Pentium-III 450MHz machine (128MB RAM, ref: next chapter), running on LINUX platform. The codes used to generate problem instances (and corresponding domains) can be found in the appendix. The time shown in the tables below are in seconds. Four algorithms were used: Graphplan , WALKSAT, SATZ, and Randomized version of SATZ. If no plan was found in about 10 minutes, then the program was treminated, and restarted again upto five times (the problem generating code is seeded. So each run generates a different instance). Empty cells denote that no plan was found even after 5 trials. The computer was left alone while the experiments were running.

Sentence Size	Varsize: 30			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	1.28	1.26	1.26	1.23
40	2.72	2.70	2.69	2.68
60	5.90	5.34	5.41	4.83
80	56.20	8.86	10.88	72.00
100	236.32	227.26	221.00	156.45

Table 3.2: Time taken by BLACKBOX to find a plan for problems containing 30 variables

Sentence Size	Varsize: 40			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	3.35	3.30	3.34	3.31
40	6.49	6.49	6.49	6.45
60	111.37	128.59	131.99	86.50
80	270.90	274.40	271.90	—
100	—	—	—	—

Table 3.3: Time taken by BLACKBOX to find a plan for problems containing 40 variables

Sentence Size	Varsize: 50			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	7.62	7.56	7.55	7.49
40	159.41	365.90	377.00	167.18
60	—	—	—	—

Table 3.4: Time taken by BLACKBOX to find a plan for problems containing 50 variables

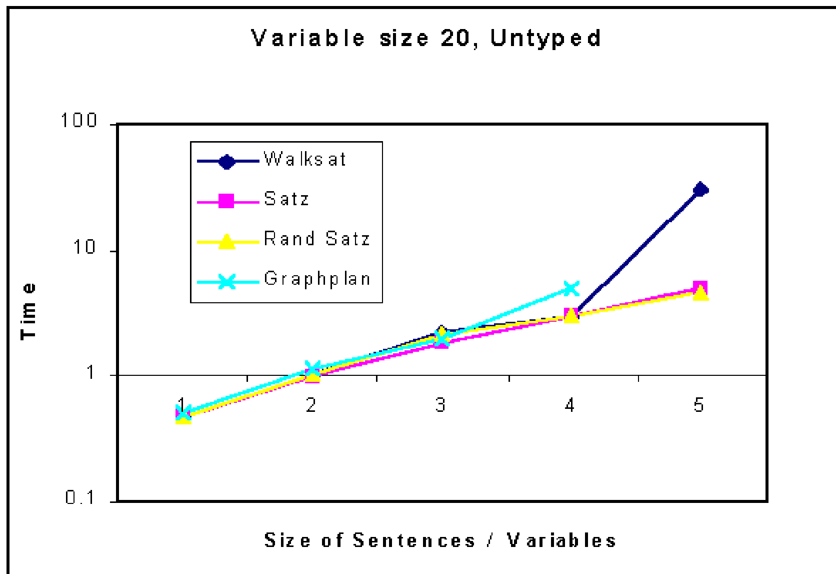

Figure 3.1: Plot of the time taken (on logarithmic scale) to find a plan for problems containing 20 variables vs. the ratio of the number of sentences and the number of variables

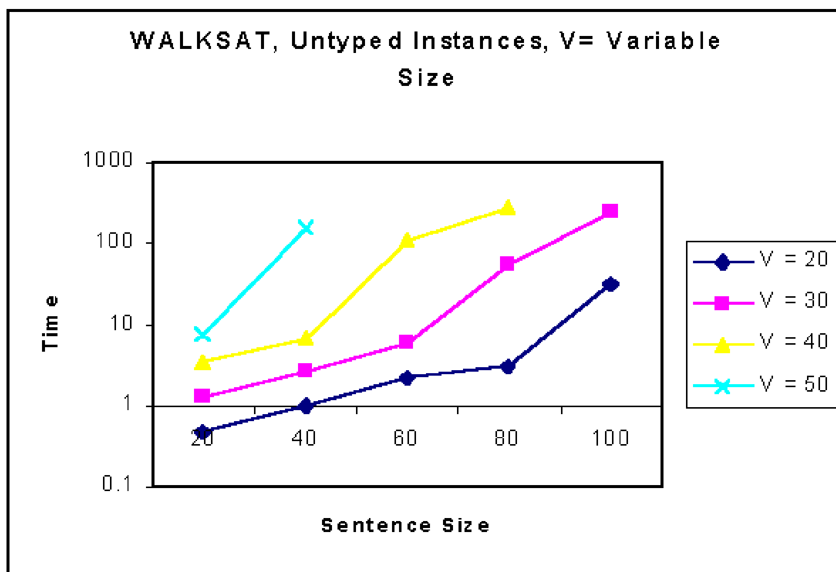

Figure 3.2: Plot of the time taken by WALKSAT to find a plan vs. the number of sentences. The number of variables is the parameter

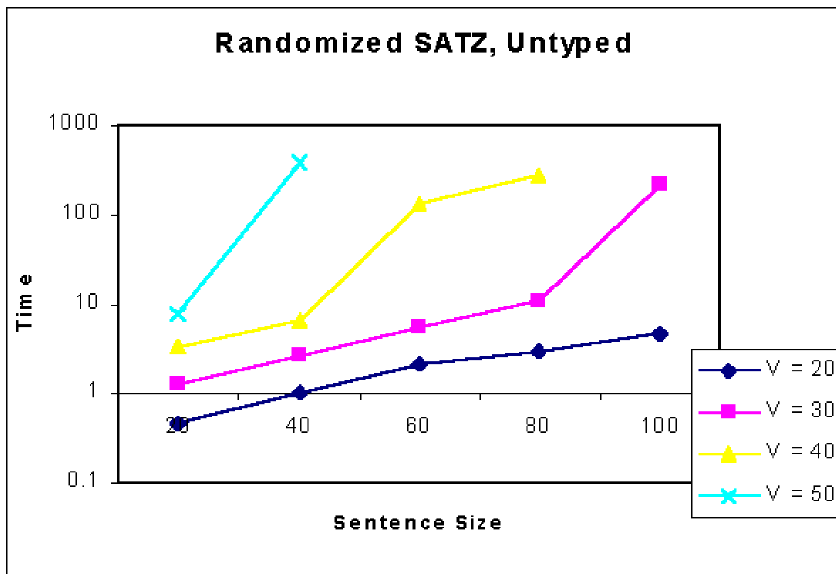

Figure 3.3: Plot of the time taken by randomized SATZ (the data for SATZ is similar) to find a plan vs. the number of sentences. The number of variables is the parameter

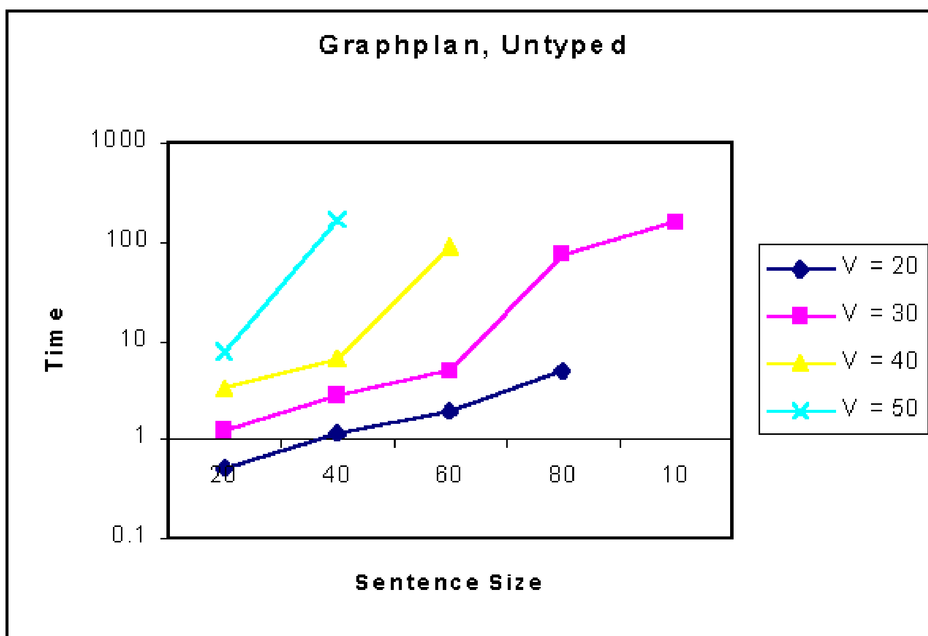

Figure 3.4: Plot of the time taken by Graphplan to find a plan vs. the number of sentences. The number of variables is the parameter

3.3 Including Typing

Typing information can be incorporated in the coding of the problem. The variable types can be separated from value types as follows:

```
;; Problem file
(:objects  A, B, C, ...
goal1, goal2, goal3, ...
unbound, true, false)
(:init  (VAROBJ A )     ;;; A, B, C etc. are VAROBJs
(VAROBJ B )
...
(VARVAL true)   ;;; true and false are VARVALs
(VARVAL false)
(value A unbound)
(value B unbound)
...
(value goal1 false)
(value goal2 false)
... )
(:goal (and  (value goal1 true)
(value goal2 true)
... ))

;; Domain file
(:action SET_VAR
```

```
 :parameters
(?var ?value)
 :precondition
(and  (VAROBJ ?var)     ;;; ?var is of type VAROBJ
  (VARVAL ?value)
(value ?var unbound))
:effect
(and  (value ?var ?value)
  (not (value ?var unbound ))))
(:action OR1
 :precondition
(value A true )
:effect
(and  (value goal1 true )
(not (value goal1 false ))))
...
```

Sentence Size	Varsize: 20			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.05	0.03	0.03	0.03
40	0.06	0.05	0.05	0.04
60	0.07	0.06	0.06	0.11
80	0.12	0.08	0.08	596.28
100	0.15	0.09	0.10	—

Table 3.5: Time taken by BLACKBOX to find a plan for problems containing 20 variables and typing

Sentence Size	Varsize: 40			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.07	0.05	0.05	0.05
40	0.08	0.07	0.07	0.06
60	0.09	0.08	0.08	0.07
80	0.10	0.09	0.09	4595.17
100	0.11	0.11	0.11	—

Table 3.6: Time taken by BLACKBOX to find a plan for problems containing 40 variables and typing

Sentence Size	Varsize: 60			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.09	0.08	0.08	0.07
40	0.10	0.09	0.09	0.08
60	0.11	0.10	0.10	0.09
80	0.12	0.12	0.12	0.16
100	0.13	0.13	0.13	0.18

Table 3.7: Time taken by BLACKBOX to find a plan for problems containing 60 variables and typing

Sentence Size	Varsize: 80			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.12	0.10	0.10	0.12
40	0.13	0.12	0.11	0.15
60	0.14	0.13	0.13	0.17
80	0.15	0.14	0.14	0.19
100	0.16	0.16	0.16	0.19

Table 3.8: Time taken by BLACKBOX to find a plan for problems containing 80 variables and typing

	Varsize: 100			
Sentence Size	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.14	0.13	0.13	0.15
40	0.15	0.14	0.14	0.14
60	0.16	0.16	0.16	0.16
80	0.17	0.17	0.17	0.21
100	0.19	0.19	0.19	0.24

Table 3.9: Time taken by BLACKBOX to find a plan for problems containing 100 variables and typing

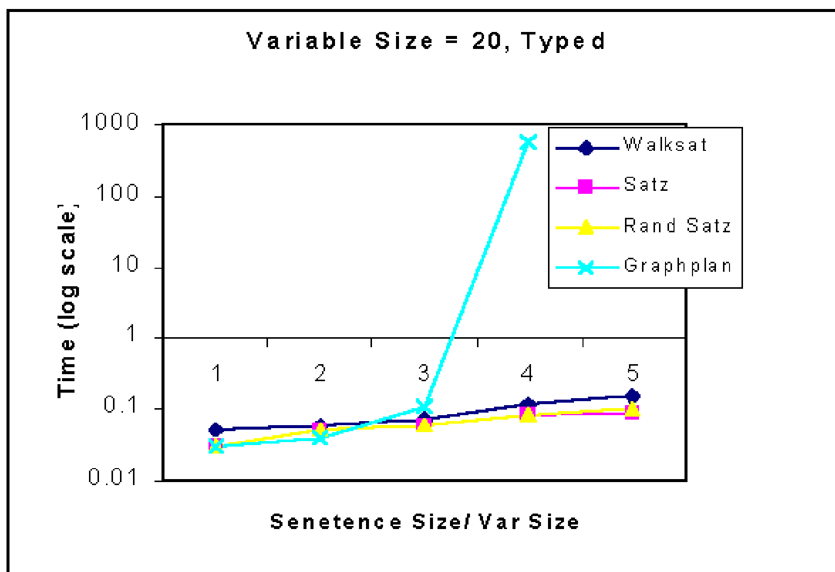

Figure 3.5: Plot of the time taken (on logarithmic scale) to find a plan for typed problems containing 20 variables vs. the ratio of the number of sentences and the number of variables

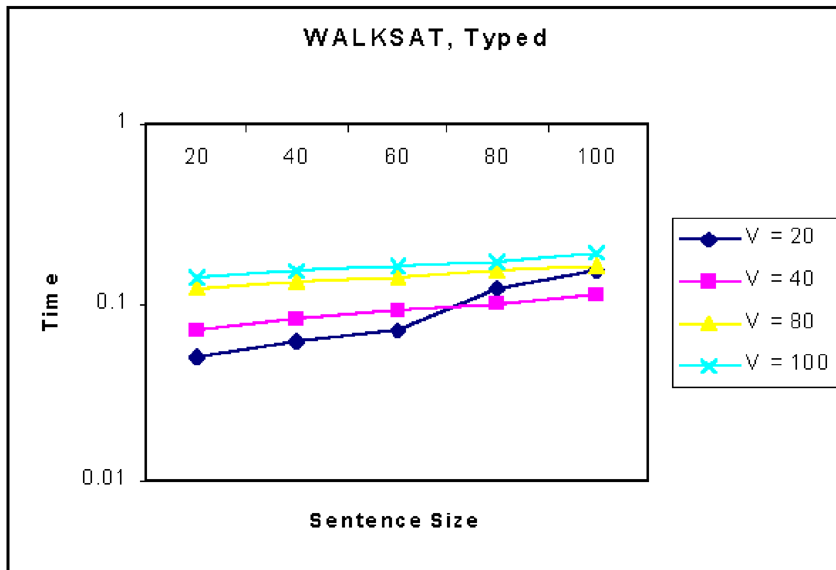

Figure 3.6: Plot of the time taken by WALKSAT (the data for SATZ, and RAND-SATZ is similar) to find a plan vs. the number of sentences. The number of variables is the parameter

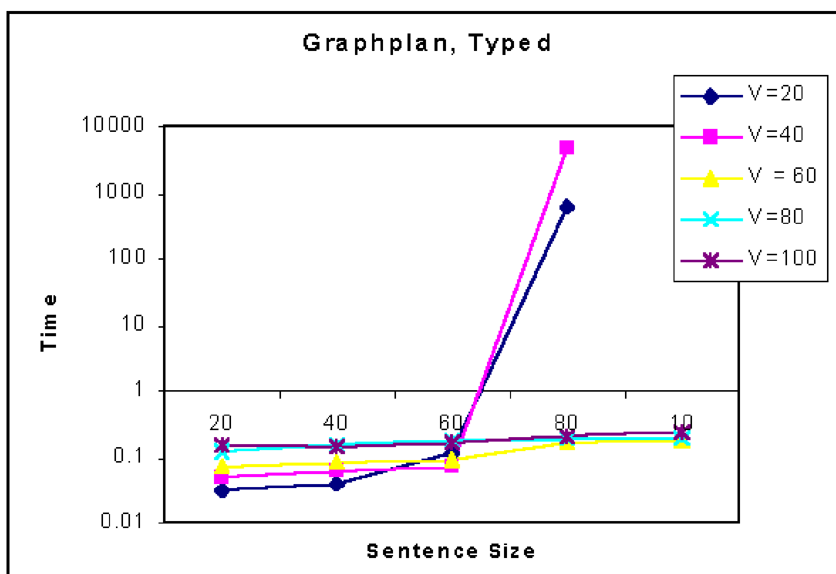

Figure 3.7: Plot of the time taken by Graphplan to find a plan for typed problems vs. the number of sentences. The number of variables is the parameter

3.4 Carrying On

In this third part of the experiment, all the variables were initialized as true, and there was one operator that changed the value from true to false. Carrying on the "Satisfiability as Planning" (section 3.1) analogy, this experiment is truly a satsfiability problem, posed as a planning problem.

```
(:init  (VAROBJ A )     ;;; A, B, C etc. are VAROBJs
(VAROBJ B )
...
(VARVAL true)   ;;; true and false are VARVALs
(VARVAL false)
(value A true)  ;;; all variables are initialized to true
(value B true)
...
(value goal1 false) ...)

(:action SET_FALSE    ;;SET-VAR operator of previous sections
 :parameters
(?var)
 :precondition
(and  (VAROBJ ?var)
  (value ?var true))
:effect
(and  (value ?var false)
  (not (value ?var true ))))
```

Sentence Size	Varsize: 20			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.04	0.03	0.03	0.03
40	0.06	0.05	0.05	1.78
100	120.14	0.29	0.28	—
200	120.29	0.71	0.70	—
400	120.70	2.75	2.67	—
500	120.90	4.61	4.47	—

Table 3.10: Time taken by BLACKBOX to find a plan for problems containing 20 variables and typing, variables initialized to true

Sentence Size	Varsize: 40			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.05	0.04	0.05	0.05
40	0.07	0.07	0.07	0.05
100	0.11	0.13	0.13	—
200	122.4	0.74	0.73	—
400	121.2	2.03	2.06	—
500	120.8	3.31	3.10	—

Table 3.11: Time taken by BLACKBOX to find a plan for problems containing 40 variables and typing, variables initialized to true

Sentence Size	Varsize: 100			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.10	0.09	0.09	0.10
40	0.11	0.10	0.11	0.12
100	0.15	0.18	0.18	0.15
200	0.23	0.31	0.33	—
400	120.62	3.54	58.12	—
500	120.82	3.83	907.9	—

Table 3.12: Time taken by BLACKBOX to find a plan for problems containing 100 variables and typing, variables initialized to true

Sentence Size	Varsize: 200			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.17	0.16	0.16	0.17
40	0.19	0.18	0.18	0.19
100	0.23	0.27	0.27	0.25
200	0.32	0.43	0.43	—
400	0.58	0.97	0.98	—
500	0.74	1.35	1.22	—

Table 3.13: Time taken by BLACKBOX to find a plan for problems containing 200 variables and typing, variables initialized to true

Sentence Size	Varsize: 300			
	Walksat	SATZ	RAND-SATZ	Graphplan
20	0.25	0.24	0.24	0.25
40	0.27	0.27	0.27	0.27
100	0.33	0.39	0.37	0.35
200	0.44	0.57	0.57	0.49
400	0.81	1.26	1.19	—
500	1.05	1.60	1.61	—

Table 3.14: Time taken by BLACKBOX to find a plan for problems containing 300 variables and typing, variables initialized to true

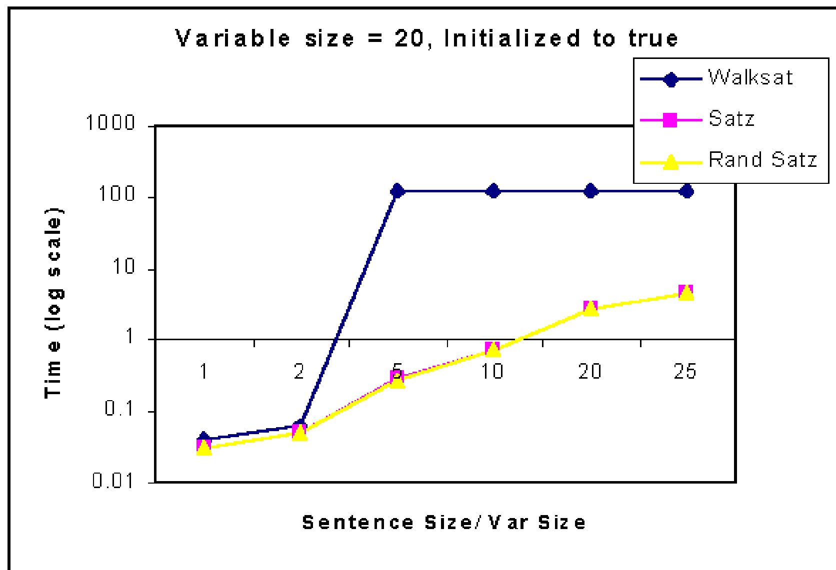

Figure 3.8: Plot of the time taken to find a plan for typed problems containing 20 variables vs. the ratio of the number of sentences and the number of variables (initialized to true)

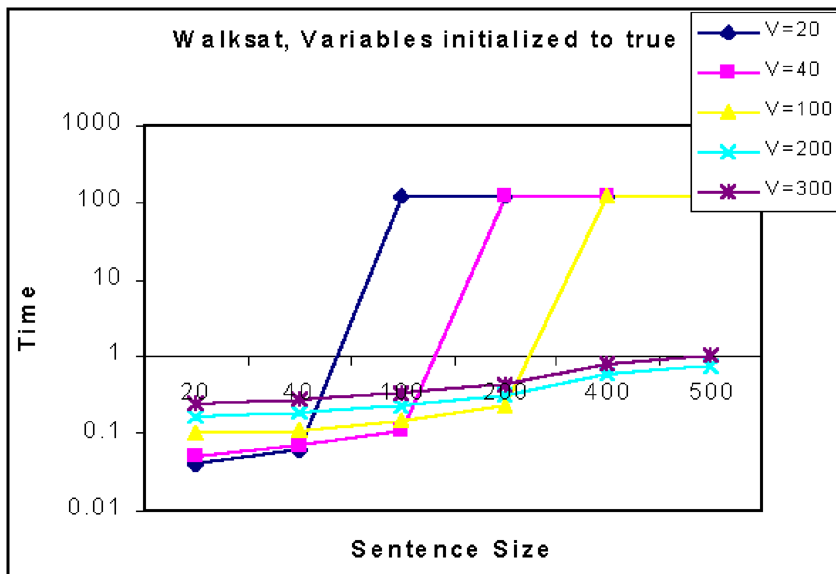

Figure 3.9: Plot of the time taken by WALKSAT to find a plan vs. the number of sentences. The number of variables (initialized to true) is the parameter

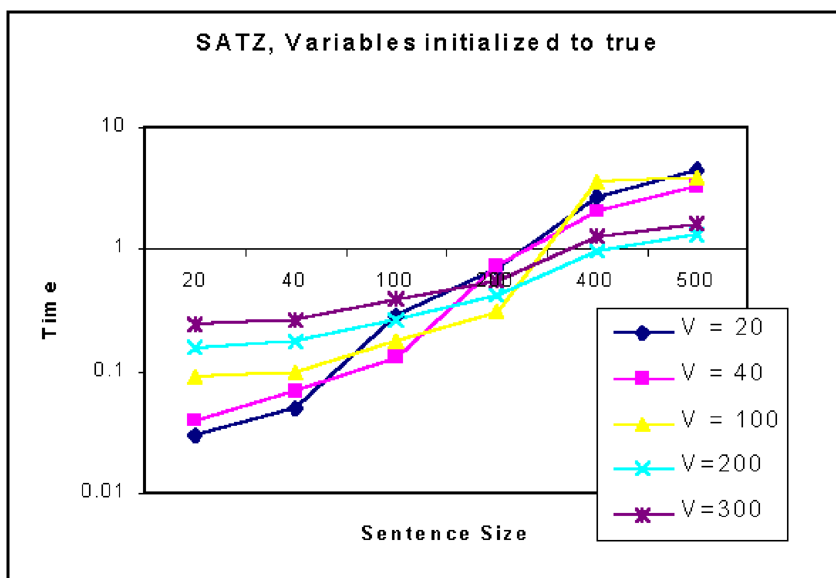

Figure 3.10: Plot of the time taken by SATZ to find a plan vs. the number of sentences. The number of variables (initialized to true) is the parameter

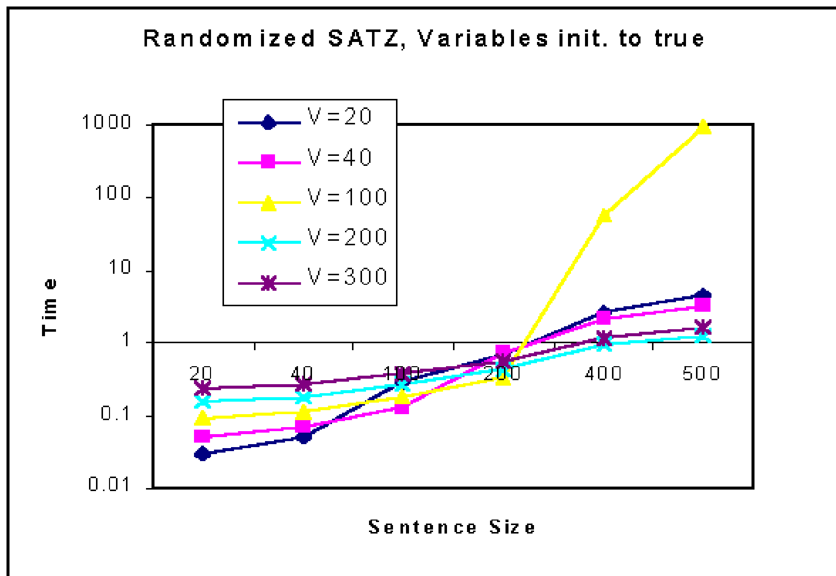

Figure 3.11: Plot of the time taken by RAND-SATZ to find a plan vs. the number of sentences. The number of variables (initialized to true) is the parameter

And finally, though no change to the previous results is anticipated, but just for the sake of curiosity, one more operator was introduced that had the effect of flipping the effect of the first operator, SET-FALSE. Interestingly, the second operator was never invoked, so the results remained the same as in the previous experiment. The additional operator was

```
;;; Change the value from false to true
(:action SET_TRUE
 :parameters
(?var)
 :precondition
(and  (VAROBJ ?var) (value ?var false))
:effect
(and  (value ?var true)
  (not (value ?var false ))))
```

Chapter 4

Conclusion and Further Study

The BLACKBOX planning system used RAM rather heavily, and therefore its performance was very much RAM dependant. Oftentimes while running a test, the computer would bcome totally paralized. I removed 64MB from my computer and ran the supplied `log.d` problem using randomized SATZ as solver. It took 438 seconds. But when I ran it using 128MB in memory slots, it took 13 seconds on the average. Further, on problems of small size, for example, `rocket.a` there was no difference in timing - in each configuration, it took 7.8 seconds on average. I also found that some of the large problems that were not solved with 64MB configuration for even for 2-3 hours, were solved with 128MB configuration. One further work could be to modify the system to reduce its heavy memory dependancy. Including typing information in the problem reduced the search time cosiderably, and the system was able to solve larger problems.

In my experiments the number of oprators was proportional to the size of the problem (the number of sentences to be precise). If S is the number of 3-CNF sentences present in the goal, then the number of operators equal $3S + 1$. This (large number of operators) was the source of “hradness” of the problem. Stochastic solver algorithms, WALKSAT, and randomized SATZ took same or more time as compared

to systematic solver SATZ. For small problems all four algorithms - WALKSAT, SATZ, Rand-SATZ, and Graphplan took comparable time, but WALKSAT and Graphplan performed worse on larger problems. Graphplan failed to find a plan for problems of large size. A rather large operator set is the more likely reason. There is very strong correlation between the smallness of time and Graphplan 's success in finding a plan i.e. either it found the plan very quickly, or it failed (or took enormous amount of time).

The fraction of time taken by Graphplan , the front end of BLACKBOX, seems to be huge as compared to the time taken by the SAT solvers. The exponential behavior present in the graphs is probably coming from Graphplan rather than the algorithms themselves. I encoded some of the experiments in SAT directly, and ran GSAT (the code was written by Kautz and Selman) on them. GSAT solved them virtually instantaneously. Even the largest problem of the prevoius chapter took a negligible fraction of a millisecond.

Chapter 5

Appendix

5.1 Code for Generating Untyped Problem Instances

```
// File: gentyped.c
// Author: Shashi Prabh
// Compile: cc genuntyped.c -o genuntyped
// Usage genuntyped [#vars, default 10] [#sentences, default 20][cnf_size, defualt 3]
// Example: genuntyped 20 40 3

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
 FILE *stream;

int main (int argc, char *argv[])
```

```
{
int varsize, sentsize,n_cnf,i,j;
char file[20];
char dname[20];
char pname[20];
srand( (unsigned)time( NULL ) );

if (argc >1)
varsize = atoi(argv[1]);
else
varsize = 10;
if (argc >2)
sentsize = atoi(argv[2]);
else
sentsize = 20;
if (argc >3)
n_cnf = atoi(argv[3]);
else
n_cnf = 3;

strcpy( file, "domain" );
if (argc >1)
strcat( file, argv[1]);
else
strcat( file, "10");
strcat( file, "_" );
if (argc >2)
```

```
strcat( file, argv[2]);
else
strcat( file, "100");
strcpy(dname, file );
strcat( file, ".pddl" );


stream = freopen( file, "w", stderr );

if( stream == NULL )
fprintf( stdout, "error on freopen\n" );
else
{
fprintf( stream, ";; domain file\n" );
fprintf( stream, ";;# of Variables %i, # of sentences %i \n", varsize, sentsize);
fprintf( stream, "(define (domain " );
fprintf( stream, dname );
fprintf( stream, ") \n" );
fprintf( stream, "\t(:requirements :strips) \n" );
fprintf( stream, "\t(:predicates (value ?x ?y))\n" );
fprintf( stream, "(:action SET_VAR \n" );
fprintf( stream, "\t :parameters \n" );
fprintf( stream, "\t\t(?var ?value)\n" );
fprintf( stream, "\t :precondition \n" );
fprintf( stream, "\t\t (value ?var unbound)\n" );
fprintf( stream, "\t:effect \n" );
fprintf( stream, "\t\t (and (value ?var ?value)\n" );
```

```
fprintf( stream, "\t\t (not (value ?var unbound ))))\n" );
// action set_var
for (i=1; i <= sentsize; i++)
{
for (j=1; j <= n_cnf; j++)
{
fprintf( stream, "(:action OR" );
fprintf( stream, "%i_%i\n",i,j);
fprintf( stream, "\t :precondition \n" );
fprintf( stream, "\t\t (value var" );
fprintf( stream, "%i ",rand()%varsize +1);
fprintf( stream, "%s )\n",rand()%2 ? "false" : "true");
fprintf( stream, "\t:effect \n" );
fprintf( stream, "\t\t(and (value goal" );
fprintf( stream, "%i ",i);
fprintf( stream, "true )\n");
fprintf( stream, "\t\t (not (value goal" );
fprintf( stream, "%i ",i);
fprintf( stream, "false ))))\n");
}
}

fprintf( stream, ")\n");

fprintf( stdout, "Domain file created\n" );
fclose( stream );
}
```

```
//Domain file though no such distiction exist here...

strcpy( file, "prob" );
if (argc >1)
strcat( file, argv[1]);
else
strcat( file, "10");
strcat( file, "_" );
if (argc >2)
strcat( file, argv[2]);
else
strcat( file, "100");
strcpy(pname, file );
strcat( file, ".pddl" );



stream = freopen( file, "w", stderr );

if( stream == NULL )
fprintf( stdout, "error on freopen\n" );
else
{
fprintf( stream, ";; problem file\n" );
fprintf( stream, ";;# of Variables %i, # of sentences %i \n", varsize, sentsize);
fprintf( stream, "(define (problem " );
fprintf( stream, pname );
fprintf( stream, ") \n" );
```

```
fprintf( stream, "\t(:domain " );
fprintf( stream, dname );
fprintf( stream, ") \n" );

fprintf( stream, "\t(:objects \n" );
for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t var" );
fprintf( stream, "%i\n", i);
}
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t goal" );
fprintf( stream, "%i\n", i);
}
fprintf( stream, "\t\t true \n\t\t false \n\t\t unbound )\n" );
// objects

fprintf( stream, "\t(:init \n" );
for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t (value var" );
fprintf( stream, "%i unbound)\n", i);
}
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t (value goal" );
```

```
fprintf( stream, "%i false)\n", i);
}
fprintf( stream, "\t)\n" );

fprintf( stream, "\t(:goal \n \t\t (and \n" );
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t (value goal" );
fprintf( stream, "%i true)\n", i);
}
fprintf( stream, "\t)))\n" );

}
//Problem file
fprintf( stdout, "Problem file created\n" );
fclose( stream );

return 0;
}
```

5.2 Code for Generating Typed Problem Instances

```
// File: gentyped.c
// Author: Shashi Prabh
// Compile: cc gentyped.c -o gentyped
// Usage gentyped [#vars, default 10] [#sentences, default 20][cnf_size, defualt 3]
```

```
// Example: gentyped 20 40 3
// typed
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
 FILE *stream;

int main (int argc, char *argv[])
{
int varsize, sentsize,n_cnf,i,j;
char file[20];
char dname[20];
char pname[20];
srand( (unsigned)time( NULL ) );

if (argc >1)
varsize = atoi(argv[1]);
else
varsize = 10;
if (argc >2)
sentsize = atoi(argv[2]);
else
sentsize = 20;
if (argc >3)
n_cnf = atoi(argv[3]);
else
```

```
n_cnf = 3;

strcpy( file, "domain" );
if (argc >1)
strcat( file, argv[1]);
else
strcat( file, "10");
strcat( file, "_" );
if (argc >2)
strcat( file, argv[2]);
else
strcat( file, "100");
strcpy(dname, file );
strcat( file, ".pddl" );



stream = freopen( file, "w", stderr );

if( stream == NULL )
fprintf( stdout, "error on freopen\n" );
else
{
fprintf( stream, ";; domain file\n" );
fprintf( stream, ";;# of Variables %i, # of sentences %i \n", varsize, sentsize);
fprintf( stream, "(define (domain " );
fprintf( stream, dname );
fprintf( stream, ") \n" );
```

```
fprintf( stream, "\t(:requirements :strips) \n" );
fprintf( stream, "\t(:predicates (value ?x ?y))\n" );
fprintf( stream, "(:action SET_VAR \n" );
fprintf( stream, "\t :parameters \n" );
fprintf( stream, "\t\t(?var ?value)\n" );
fprintf( stream, "\t :precondition \n" );
fprintf( stream, "\t\t (and (VAROBJ ?var) \n \t\t");
fprintf( stream, "(VARVAL ?value)(value ?var unbound))\n" );
fprintf( stream, "\t:effect \n" );
fprintf( stream, "\t\t (and (value ?var ?value)\n" );
fprintf( stream, "\t\t (not (value ?var unbound ))))\n" );
// action set_var
for (i=1; i <= sentsize; i++)
{
for (j=1; j <= n_cnf; j++)
{
fprintf( stream, "(:action OR" );
fprintf( stream, "%i_%i\n",i,j);
fprintf( stream, "\t :precondition \n" );
fprintf( stream, "\t\t (value var" );
fprintf( stream, "%i ",rand()%varsize +1);
fprintf( stream, "%s )\n",rand()%2 ? "false" : "true");
fprintf( stream, "\t:effect \n" );
fprintf( stream, "\t\t(and (value goal" );
fprintf( stream, "%i ",i);
fprintf( stream, "true )\n");
fprintf( stream, "\t\t (not (value goal" );
```

```
fprintf( stream, "%i ",i);
fprintf( stream, "false ))))\n");
}
}

fprintf( stream, ")\n");

fprintf( stdout, "Domain file created\n" );
fclose( stream );
}
//Domain file though no such distiction exist here...

strcpy( file, "prob" );
if (argc >1)
strcat( file, argv[1]);
else
strcat( file, "10");
strcat( file, "_" );
if (argc >2)
strcat( file, argv[2]);
else
strcat( file, "100");
strcpy(pname, file );
strcat( file, ".pddl" );


stream = freopen( file, "w", stderr );
```

```
if( stream == NULL )
fprintf( stdout, "error on freopen\n" );
else
{
fprintf( stream, ";; problem file\n" );
fprintf( stream, ";;# of Variables %i, # of sentences %i \n", varsize, sentsize);
fprintf( stream, "(define (problem " );
fprintf( stream, pname );
fprintf( stream, ") \n" );
fprintf( stream, "\t(:domain " );
fprintf( stream, dname );
fprintf( stream, ") \n" );

fprintf( stream, "\t(:objects \n" );
for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t var" );
fprintf( stream, "%i\n", i);
}
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t goal" );
fprintf( stream, "%i\n", i);
}
fprintf( stream, "\t\t true \n\t\t false )\n" );
// objects
```

```
fprintf( stream, "\t(:init \n" );

for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t (VAROBJ var" );
fprintf( stream, "%i)\n", i);
}

fprintf( stream, "\t\t (VARVAL true) \n" );
fprintf( stream, "\t\t (VARVAL false) \n" );


for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t (value var" );
fprintf( stream, "%i unbound)\n", i);
}
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t (value goal" );
fprintf( stream, "%i false)\n", i);
}
fprintf( stream, "\t)\n" );

fprintf( stream, "\t(:goal \n \t\t (and \n" );
for (i=1; i <= sentsize; i++)
```

```
{
fprintf( stream, "\t\t (value goal" );
fprintf( stream, "%i true)\n", i);
}
fprintf( stream, "\t)))\n" );

}
//Problem file
fprintf( stdout, "Problem file created\n" );
fclose( stream );

return 0;
}
```

5.3 Code for Generating SAT-Like Problem Instances

```
// File: gensat.c
// Author: Shashi Prabh
// Compile: cc gensat.c -o gensat
// Usage gensat [#vars, default 10] [#sentences, default 20][cnf_size, defualt 3]
// Example: gensat 20 40 3
```

```
// expt: init: true, one op: true->false
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
 FILE *stream;

int main (int argc, char *argv[])
{
int varsize, sentsize,n_cnf,i,j;
char file[20];
char dname[20];
char pname[20];
srand( (unsigned)time( NULL ) );

if (argc >1)
varsize = atoi(argv[1]);
else
varsize = 10;
if (argc >2)
sentsize = atoi(argv[2]);
else
sentsize = 20;
if (argc >3)
n_cnf = atoi(argv[3]);
else
n_cnf = 3;
```

```
strcpy( file, "domain" );
if (argc >1)
strcat( file, argv[1]);
else
strcat( file, "10");
strcat( file, "_" );
if (argc >2)
strcat( file, argv[2]);
else
strcat( file, "100");
strcpy(dname, file );
strcat( file, ".pddl" );


stream = freopen( file, "w", stderr );

if( stream == NULL )
fprintf( stdout, "error on freopen\n" );
else
{
fprintf( stream, ";; domain file\n" );
fprintf( stream, ";;# of Variables %i, # of sentences %i \n", varsize, sentsize);
fprintf( stream, "(define (domain " );
fprintf( stream, dname );
fprintf( stream, ") \n" );
fprintf( stream, "\t(:requirements :strips) \n" );
```

```
fprintf( stream, "\t(:predicates (value ?x ?y))\n" );
fprintf( stream, "(:action SET_VAR \n" );
fprintf( stream, "\t :parameters \n" );
fprintf( stream, "\t\t(?var )\n" );
fprintf( stream, "\t :precondition \n" );
fprintf( stream, "\t\t (and (VAROBJ ?var) \n (value ?var true))\n" );
fprintf( stream, "\t:effect \n" );
fprintf( stream, "\t\t (and (value ?var false)\n" );
fprintf( stream, "\t\t (not (value ?var true ))))\n" );
// action set_var
for (i=1; i <= sentsize; i++)
{
for (j=1; j <= n_cnf; j++)
{
fprintf( stream, "(:action OR" );
fprintf( stream, "%i_%i\n",i,j);
fprintf( stream, "\t :precondition \n" );
fprintf( stream, "\t\t (value var" );
fprintf( stream, "%i ",rand()%varsize +1);
fprintf( stream, "%s )\n",rand()%2 ? "false" : "true");
fprintf( stream, "\t:effect \n" );
fprintf( stream, "\t\t(and (value goal" );
fprintf( stream, "%i ",i);
fprintf( stream, "true )\n");
fprintf( stream, "\t\t (not (value goal" );
fprintf( stream, "%i ",i);
fprintf( stream, "false ))))\n");
```

```
}
}

fprintf( stream, ")\n");

fprintf( stdout, "Domain file created\n" );
fclose( stream );
}
//Domain file though no such distiction exist here...

strcpy( file, "prob" );
if (argc >1)
strcat( file, argv[1]);
else
strcat( file, "10");
strcat( file, "_" );
if (argc >2)
strcat( file, argv[2]);
else
strcat( file, "100");
strcpy(pname, file );
strcat( file, ".pddl" );



stream = freopen( file, "w", stderr );

if( stream == NULL )
```

```
fprintf( stdout, "error on freopen\n" );
else
{
fprintf( stream, ";; problem file\n" );
fprintf( stream, ";;# of Variables %i, # of sentences %i \n", varsize, sentsize);
fprintf( stream, "(define (problem " );
fprintf( stream, pname );
fprintf( stream, ") \n" );
fprintf( stream, "\t(:domain " );
fprintf( stream, dname );
fprintf( stream, ") \n" );

fprintf( stream, "\t(:objects \n" );
for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t var" );
fprintf( stream, "%i\n", i);
}
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t goal" );
fprintf( stream, "%i\n", i);
}
fprintf( stream, "\t\t true \n\t\t false )\n" );
// objects

fprintf( stream, "\t(:init \n" );
```

```
for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t (VAROBJ var" );
fprintf( stream, "%i)\n", i);
}

//fprintf( stream, "\t\t (VARVAL true) \n" );
//fprintf( stream, "\t\t (VARVAL false) \n" );



for (i=1; i <= varsize; i++)
{
fprintf( stream, "\t\t (value var" );
fprintf( stream, "%i true)\n", i);
}
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t (value goal" );
fprintf( stream, "%i false)\n", i);
}
fprintf( stream, "\t)\n" );

fprintf( stream, "\t(:goal \n \t\t (and \n" );
for (i=1; i <= sentsize; i++)
{
fprintf( stream, "\t\t (value goal" );
```

```
fprintf( stream, "%i true)\n", i);
}
fprintf( stream, "\t)))\n" );

}
//Problem file
fprintf( stdout, "Problem file created\n" );
fclose( stream );

return 0;
}
```

BIBLIOGRAPHY

[1] Bayardo, R. Using CSP look-back techniques to solve real world SAT instances. *Proc. AAAI 97* (1997).

[2] Blum, A., and Furst, M. Fast planning through planning graph analysis. *Artififcial Intelligence 90* (1997), 281–300.

[3] Do, M., and Kambhampati, S. Solving planning-graph by compiling it into CSP. *http://rakaposhi.eas.asu.edu/papers-by-year.html* (2001).

[4] Fikes, R., and Nilsson, N. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2* (1971), 189–208.

[5] Kambhampati, S. Challenges in bridging plan synthesis paradigms. *Proc. IJCAI-97* (1997).

[6] Kautz, H., and Selman, B. Planning as satisfiability. *Proc. 10th European Conference on Artificial Intelligence (ECAI 92)* (1992), 359–363.

[7] Kautz, H., and Selman, B. Unifying SAT-based and Graph-based planning. *Proc. IJCAI 99* (1999).

[8] Kautz, H. and Walser, J. Integer optimization models of AI planning problems. *www.cs.washington.edu/homes/kautz/papers/index.html* (1999).

[9] Li, C., and Anbulagan. Heuristics based on unit propagation for satisfiability problems. *Proc. IJCAI-97* (1997).

[10] McCarthy, J., and Hayes, P. J. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4* (1969), 463–502.

[11] McDermott, D. PDDL — The Planning Domain Definition Language, version 1.2. Tech. Rep. CVC TR-98-003/ DCS TR-1165, Computer Science Department, Yale University, New Haven, CT, 1998.

[12] Nilsson, N. J. *Artificial Intelligence: A New Synthesis.* Morgan-Kaufmann, San Francisco, California, 1998.

[13] Selman, B., Levesque, H., and Mitchell, D. A new method for solving hard satisfiability problems. *Proc. AAAI92* (1992).

[14] Weld, D. Recent advances in planning. Tech. Rep. UW-CSE-98-10-01, Computer Science Department, University of Washington, Seattle, WA, 1999.

[15] Wilkins, W., and desJardins, M. A call for knowledge-based planning. *AI Magazine 22*, 1 (2001), 99–115.