Solving Non-Boolean Satisfiability Problems with the Davis-Putnam Method

Peter G. Stock

October 1999 – March 2000

Abstract

The Davis-Putnam method has received recent attention for its ability to solve large problems. Many of these problems are non-Boolean, in that they are most naturally expressed as formulas containing variables with domains of more than two values. There are two approaches to solving non-Boolean formulas — either translate the problem into a Boolean equivalent and solve this with a Boolean procedure, or use a non-Boolean procedure directly on the non-Boolean formula. This project develops a direct non-Boolean Davis-Putnam procedure and compares the results of using the two approaches.

Contents

Notation used

$=$	Equality
$\equiv$	Equivalence
$()$	Grouping parentheses
$\{a,b,c\}$, A	Sets
$\in$	Set membership
$\notin$	Set non-membership
$\subseteq$	Subset relation
$\supset$	Proper superset relation
$\cup$	Set union
$\cap$	Set intersection
$\setminus$	Set subtraction
$\neg$	Logical "not"
$\vee$	Logical "or"
$\wedge$	Logical "and"
T, F	Truth values
$\mathbf{x}$	Boolean variable
$\mathbf{X}$	Non-Boolean variable
$\mathbf{X}\in\{a,b\}$	Non-Boolean literal
$\mathbf{x} \mapsto T$	Boolean variable assignment
$\mathbf{X} \mapsto a$	Non-Boolean variable assignment

Introduction

Boolean satisfiability

The satisfiability problem (SAT) is fundamental in solving many practical problems and spans a wide range of areas of computer science, such as computer vision, scheduling, circuit design and automated reasoning. Methods for solving SAT are therefore important for the development of efficient computing systems. SAT is of particular interest in the field of theoretical computer science, because it was the first problem to be proven to be NP-complete by Stephen Cook in 1971 [2]. Even though SAT *may* be intractable (all algorithms to solve SAT currently take exponential time in the worst case), a good average case performance is sought.

Davis-Putnam

Although the Davis-Putnam method (DP) was developed nearly forty years ago (first published in 1960 [5], revised in 1962 [6]), it still remains one of the best complete procedures for determining Boolean satisfiability. The best currently available DP-based solvers do not vary significantly from the concept described in [6], although their performance has advanced significantly. Modern DP-based solvers can now routinely solve formulas containing hundreds of variables — "SATO" has solved over one hundred open quasi-group problems in design theory. Although incomplete methods have recently come into the limelight for their performance, they cannot *decide* SAT, i.e. they cannot prove unsatisfiability.

Non-Boolean problems

Many problems are non-Boolean, i.e. they are most concisely formulated in terms of variables with domains other than {True, False}. A non-Boolean formula is constructed from variables which have domains of size k. In this way, non-Boolean formulas generalise Boolean formulas.

The traditional approach to solving a non-Boolean formula is to translate it into equivalent Boolean formula and solve this with a Boolean solver. The direct approach is to modify a Boolean solver so that it operates directly on the non-Boolean formula. A previous project by a student at the university of York [15] explored the direct approach by developing a non-Boolean stochastic local search procedure (NB-walksat). The results showed that the two methods are not trivially different — the direct method significantly outperformed the translation method on a range of problems.

Non-Boolean Davis-Putnam

The success of NB-walksat led to the idea that a non-Boolean generalisation of the Davis-Putnam method may show similar results. This project develops a non-Boolean version of the Davis-Putnam procedure and empirically compares it with the translation approach. The results show that as domain size increases, the direct method performs significantly better than the translation method on a number of problems, while on others the translation approach is more efficient. The performance is analysed, the strengths and weaknesses of each of the methods are discussed and a number of possible improvements are presented.

Project outline

Chapter 2 reviews the current state of the art and discusses the topics which are relevant to the development of a non-Boolean Davis-Putnam SAT solver.

Chapter 3 introduces the non-Boolean Davis-Putnam procedure and outlines it's implementation. The empirical tests are also described.

Chapter 4 presents the experimental results and analyses them.

Chapter 5 concludes the report and suggests explanations for the observed trends. The advantages and disadvantages of the non-Boolean Davis-Putnam procedure are discussed and avenues for future development are proposed.

Review

2.1 Boolean formulas

A Boolean formula is constructed from literals (variables or constants) using the operators $\neg$, $\vee$ and $\wedge$ (the $\neg$ operator conventionally has highest precedence). There are other Boolean operators but formulas containing these may be expressed as equivalent formulas containing only $\neg$, $\vee$ and $\wedge$. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, where a clause is a disjunction of literals, each of which may either be positive (a variable or constant) or negative (a variable or constant with the $\neg$ operator applied to it). All Boolean formulas can be expressed as equivalent CNF formulas.

e.g. $\quad (\mathbf{b} \vee \mathbf{r}) \;\wedge\; (\mathbf{r} \vee \mathbf{s}) \;\wedge\; (\neg\mathbf{b} \vee \neg\mathbf{r})$

A Boolean variable can have one of two values: True or False. For any variable $\mathbf{x}$, $\mathbf{x}$ is a Boolean formula. For any Boolean formulas $\mathbf{x}$ and $\mathbf{y}$, the following are also Boolean formulas and define the semantics of the operators $\neg$ (not), $\vee$ (or) and $\wedge$ (and).

$\neg\mathbf{x}$	is True if $\mathbf{x}$ is False; otherwise it is False.
$\mathbf{x} \vee \mathbf{y}$	is True if $\mathbf{x}$ is True or $\mathbf{y}$ is True (or both $\mathbf{x}$ and $\mathbf{y}$ are True); otherwise it is False.
$\mathbf{x} \wedge \mathbf{y}$	is True if both $\mathbf{x}$ and $\mathbf{y}$ are True; otherwise it is False.

$\mathbf{x}$	$\neg\mathbf{x}$
F	***T***
T	*F*

$\mathbf{x}$	$\mathbf{y}$	$\mathbf{x} \vee \mathbf{y}$
F	*F*	*F*
F	***T***	***T***
T	*F*	***T***
T	***T***	***T***

$\mathbf{x}$	$\mathbf{y}$	$\mathbf{x} \wedge \mathbf{y}$
F	*F*	*F*
F	***T***	*F*
T	*F*	*F*
T	***T***	***T***

Truth assignments

A truth assignment is a function from variables to the set of values {True, False}, a total truth assignment is a total function (every variable from the formula is mapped to a value). A truth assignment A satisfies a formula f if and only if the result of applying A to f (replacing each variable in f with its associated value from A) is True.

2.1.1 The satisfiability problem

The satisfiability problem (SAT) is to decide for a given formula whether there exists a satisfying truth assignment.

Most automated SAT solvers operate on Boolean formulas in CNF, so there is no loss of generality in considering only formulas of this type. Indeed, it is often the case that CNF is the natural encoding of many problems[1]. Let's consider an example where 3 friends are organising a fireworks night:

> Ted wants to have either Bangers or Rockets (or both).
> Paul wants either Rockets or Sparklers (or both).
> Sally doesn't like all the noise, and doesn't want both Bangers and Rockets.

Let **b** mean Bangers are included
r mean Rockets are included
s mean Sparklers are included

Then

Ted's constraint is: $\mathbf{b} \vee \mathbf{r}$

Paul's is: $\mathbf{r} \vee \mathbf{s}$

Sally's is: $\neg(\mathbf{b} \wedge \mathbf{r}) \equiv \neg\mathbf{b} \vee \neg\mathbf{r}$ (by DeMorgan's law)

To please everybody we must satisfy $(\mathbf{b} \vee \mathbf{r}) \wedge (\mathbf{r} \vee \mathbf{s}) \wedge (\neg\mathbf{b} \vee \neg\mathbf{r})$

There are three solutions that conform to everybody's constraints. Either have only rockets, have rockets and sparklers, or have bangers and sparklers. These solutions correspond to the assignments:

$\mathbf{b} \mapsto F, \quad \mathbf{r} \mapsto T, \quad \mathbf{s} \mapsto F$
$\mathbf{b} \mapsto F, \quad \mathbf{r} \mapsto T, \quad \mathbf{s} \mapsto T$
$\mathbf{b} \mapsto T, \quad \mathbf{r} \mapsto F, \quad \mathbf{s} \mapsto T$

[1] Other options include nested and/or and DNF (Disjunctive Normal Form) where each disjunct specifies the criteria for a possible solution. Formulas in DNF are trivially solvable for SAT in linear time, but the translation to DNF can introduce massive growth in the formula (effectively enumerating all possible solutions).

b	r	s	b ∨ r	r ∨ s	¬b ∨ ¬r	(b ∨ r) ∧ (r ∨ s) ∧ (¬b ∨ ¬r)
F	*F*	*F*	*F*	*F*	***T***	*F*
F	*F*	***T***	*F*	***T***	***T***	*F*
F	***T***	*F*	***T***	***T***	***T***	***T***
F	***T***	***T***	***T***	***T***	***T***	***T***
T	*F*	*F*	***T***	*F*	***T***	*F*
T	*F*	***T***	***T***	***T***	***T***	***T***
T	***T***	*F*	***T***	***T***	*F*	*F*
T	***T***	***T***	***T***	***T***	*F*	*F*

The truth-table can be constructed to enumerate all possible solutions with brute force. This method quickly becomes very inefficient for all but the simplest problems. It requires exponential space and time in the number of variables since it examines every possible truth assignment.

2.2 The Davis-Putnam method

In 1960, Martin Davis and Hilary Putnam published an article describing an efficient way of searching the truth-assignment space. Referred to as The Davis-Putnam method (or DP[2] for short), this procedure is a systematic search through a tree of *partial* truth assignments. At each node in the tree a variable is branched on — two simpler subtrees are created by assigning the variable to True in one subtree and False in the other. DP recursively explores these subtrees, essentially performing a depth-first search through the tree.

The Davis-Putnam method was soon improved by the addition of a heuristic to choose which variable to make a branch at each node in the tree. This is an important part of the DP procedure which affects how quickly solutions and contradictions are found. Now, DP can solve fairly large problems, but has been overshadowed recently by local search methods. It has recently gained attention from the development of more sophisticated heuristics and SATO (a DP-based solver) has solved some open quasi-group problems.

DP is usually restricted to work on CNF formulas, since this makes the implementation of the simplification step more efficient. If a variable is assigned to True, all clauses containing negative occurrences of that variable have the variable removed from them and all clauses containing positive occurrences of that variable are removed from the formula (vice-versa for False). If the resulting formula is empty, it has been satisfied by the current truth assignment (and any total truth assignment that extends it also satisfies the formula). If an empty clause[3] is found then an unsatisfiable state has been reached and the procedure backtracks to the last choice point in the tree and explores the other branch. If all branches have been explored then the formula is unsatisfiable.

Certain simplification steps may be applied at each node in the search tree, such as unit propagation and pure literal elimination, both described in the original Davis-Putnam article. Both techniques fix the values of variables which would otherwise be considered choice points. The unit propagation rule states that if a clause contains only a single variable, it may immediately be assigned the value that satisfies that clause (since assigning the opposite value would falsify the clause and also the formula).

[2]It is more apt to refer to the method in Loveland's form, which introduced the splitting rule in [6].

[3]An empty clause is one with no literals, which is False under all assignments because it can never contain a True literal.

e.g. $\quad (\neg \mathbf{x} \vee \neg \mathbf{y} \vee \mathbf{z}) \;\wedge\; \mathbf{y}$

$$\mathbf{y} \mapsto T$$

$$\Rightarrow \quad (\neg \mathbf{x} \vee F \vee \mathbf{z}) \;\wedge\; T$$

$$\Rightarrow \quad (\neg \mathbf{x} \vee \mathbf{z})$$

Pure literal elimination searches the formula for literals that only appear positively or only negatively. They can be assigned a value immediately, eliminating a potential choice point. It should be noted that this technique commits to branches whose alternative may not lead to immediate failure (as in unit propagation). This gives the intuitive impression of being a particularly powerful technique, although in practice pure literals do not occur frequently enough in the search to produce significant efficiency gains (see [9]).

e.g. $\quad (\mathbf{x} \vee \mathbf{y} \vee \neg \mathbf{z}) \;\wedge\; (\mathbf{x} \vee \neg \mathbf{y}) \;\wedge\; (\mathbf{y} \vee \mathbf{z})$

$$\mathbf{x} \mapsto T$$

$$\Rightarrow \quad (T \vee \mathbf{y} \vee \neg \mathbf{z}) \;\wedge\; (T \vee \neg \mathbf{y}) \;\wedge\; (\mathbf{y} \vee \mathbf{z})$$

$$\Rightarrow \quad (\mathbf{y} \vee \mathbf{z})$$

2.2.1 Definition of the Davis-Putnam method

$$\text{Definition:} \quad \begin{array}{ll} \bar{l} = \neg x & \text{if } l \text{ is } x \text{ (i.e. positive)} \\ \bar{l} = x & \text{if } l \text{ is } \neg x \text{ (i.e. negative)} \end{array}$$

```
function Davis-Putnam(formula f) return Boolean is
    /* unit propagation */
    repeat
        for each unit clause l in f do
            remove from f all clauses that contain l
            remove l̄ from all clauses in f in which it appears
        end for
        if f is empty then
            return True
        else if f contains the empty clause then
            return False
        end if
    until there are no unit clauses in f

    /* pure literal elimination */
    for all variables x that only appear positively in f do
        remove from f all clauses that contain x
    end for
    for all variables x that only appear negatively in f do
        remove from f all clauses that contain ¬x
    end for

    /* branching */
    pick a variable x from f
    if Davis-Putnam(f ∧ x) or Davis-Putnam(f ∧ ¬x) then
        return True
    else
        return False
    end if
end function
```

Note that this definition does not dictate the order of variables chosen for branching (the variable order), or the method of exploring the branches of the search tree. It is common to use a depth-first method, presenting a choice of which branch to explore first (the value order). These topics are examined a little further on.

This procedure determines whether or not a formula is satisfiable; by keeping track of the current assignment, the procedure can return a satisfying assignment if the formula is satisfiable.

Mathematical definition of unit propagation

For any CNF formula f_n and variable assignment A_n, the process of unit propagation on (f_n, A_n) produces (f_{n+1}, A_{n+1}), where

$A_{n+1} = A_n \bigcup \{l \mapsto val(l) \mid l \in U\}$
$f_{n+1} = simplify(f_n, A_{n+1})$

where U is the set of all literals that appear in unit clauses in f_n
$val(l)$ is True if l appears positively in U, False otherwise
$simplify(f, A)$ returns the simplified formula arising from applying the substitutions in A to f

For all satisfying assignments A_{sat}, if A_{sat} extends A_n, A_{sat} also extends A_{n+1}.

Mathematical definition of pure literal elimination

For any CNF formula $f_m$and variable assignment A_m, the process of pure literal elimination on (f_m, A_m) produces (f_{m+1}, A_{m+1}), where

$A_{m+1} = A_m \bigcup \{l \mapsto val(l) \mid l \in P\}$
$f_{m+1} = simplify(f_m, A_{m+1})$

where P is the set of all pure literals that appear in f_m (either all positive or all negative)
$val(l)$ is True if l appears positively in P, False otherwise
$simplify(f, A)$ returns the simplified formula arising from applying the substitutions in A to f

If there exists a satisfying assignments A_{sat} that extends A_m, then there exists a satisfying assignment A'_{sat} that extends A_{m+1} (A_{sat} and A'_{sat} may be the same). In other words, if there exists a satisfying truth assignment that extends the current variable assignment, there must also exist a satisfying assignment after the process of pure literal elimination, although the process may restrict the search, pruning some branches which contain satisfying assignments.

Both unit propagation and pure literal elimination are complete for the problem of finding a single satisfying variable assignment (given a satisfiable state[4], they do not produce an unsatisfiable state). While unit propagation is complete for the problem of finding all solutions, pure literal elimination is not, since it may eliminate areas of the search space which contain more general solutions. Most SAT solvers interpret the satisfiability problem as determining if there exists a satisfying assignment and if so returning a single solution.

[4]A satisfiable state is a state in the search from which it is possible to reach a solution. An unsatisfiable state is one where it is impossible to reach a solution from.

2.2.2 An example of the Davis-Putnam procedure

Taking our previous fireworks example, we have the following start formula:

$f_0 = (\mathbf{b} \lor \mathbf{r}) \ \land \ (\mathbf{r} \lor \mathbf{s}) \ \land \ (\neg\mathbf{b} \lor \neg\mathbf{r})$ $\qquad A_0 = \{\}$

There are no unit clauses so unit propagation can't take place. Suppose $\mathbf{r}$ is chosen to branch on. Then we have the choices $\mathbf{r} \mapsto T$ and $\mathbf{r} \mapsto F$. Suppose $\mathbf{r} \mapsto T$ is explored first.

$\mathbf{r} \mapsto T$

$f_1 = (\mathbf{b} \lor T) \ \land \ (T \lor \mathbf{s}) \ \land \ (\neg\mathbf{b} \lor F)$

$f_1 = \neg\mathbf{b}$ $\qquad A_1 = \{\mathbf{r} \mapsto T\}$

Unit propagation is now applied to make the assignment $\mathbf{b} \mapsto F$.

$\mathbf{b} \mapsto F$

$f_2 = T$

$f_2 =$ *empty formula* $\qquad A_2 = \{\mathbf{b} \mapsto F, \mathbf{r} \mapsto T\}$

The empty formula has been found and the formula has been satisfied. Note that it has been satisfied with the non-total assignment A_2 (S has not been given a value). This means that any total assignment that extends A_2 will satisfy f_0. The two total assignments are:

$\{\mathbf{b} \mapsto F, \mathbf{r} \mapsto T, \mathbf{s} \mapsto T\}$
$\{\mathbf{b} \mapsto F, \mathbf{r} \mapsto T, \mathbf{s} \mapsto F\}$

Suppose we had chosen $\mathbf{r} \mapsto F$ as the first branch to explore, instead of $\mathbf{r} \mapsto T$. Then the search would have discovered the other solution from our truth table.

$\mathbf{r} \mapsto F$

$f_1 = (\mathbf{b} \lor F) \ \land \ (F \lor \mathbf{s}) \ \land \ (\neg\mathbf{b} \lor T)$

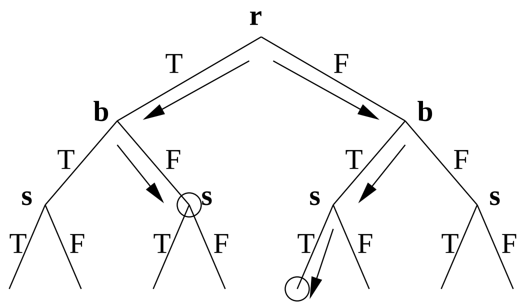
$f_1 = \mathbf{b} \ \land \ \mathbf{s}$ $\qquad A_1 = \{\mathbf{r} \mapsto F\}$

Unit propagation forces $\mathbf{b} \mapsto T$ and $\mathbf{s} \mapsto T$.

$\mathbf{b} \mapsto T \quad \mathbf{s} \mapsto T$

$f_2 = T \ \land \ T$

$f_2 =$ *empty formula* $\qquad A_2 = \{\mathbf{b} \mapsto T, \mathbf{r} \mapsto F, \mathbf{s} \mapsto T\}$



The search tree above shows both solution points. Note that the procedure does not need to explore the whole tree and that backtracking was unnecessary in this instance.

2.2.3 Variable order

The large size of the potential search space is the main problem with the DP procedure, but choosing the best variable to branch on at each node in the search tree can greatly reduce the size of the explored search space. The variable order dictates which variable is chosen to branch on at each node in the search tree and has a dramatic effect on the efficiency of the Davis-Putnam procedure. The branching heuristic decides the variable order. There are a number of choices of heuristic, both static (where the complete variable ordering is determined before the search begins) and dynamic (which identify the next variable to branch on at each node in the search tree). Most modern solvers use dynamic heuristics — the reduction in search tree size far outweighs the computational cost of applying the heuristic at each node.

The simple moms heuristic

A popular heuristic is to choose a variable[5] that has Maximal Occurrences in Minimal Size clauses (the mom's heuristic). Intuitively it chooses the most constrained variable to branch on, increasing the effect of unit propagation after the branch.

e.g. $f = (\mathbf{a} \vee \mathbf{e}) \;\wedge\; (\neg\mathbf{a} \vee \mathbf{b}) \;\wedge\; (\neg\mathbf{a} \vee \mathbf{c}) \;\wedge\; (\neg\mathbf{b} \vee \neg\mathbf{d}) \;\wedge\; (\mathbf{b} \vee \neg\mathbf{c} \vee \mathbf{d})$

Variable	Occurrences in clauses of length 2	positive	negative	Occurrences in clauses of length 3	positive	negative
a	**3**	1	2	**0**	0	0
b	**2**	1	1	**1**	1	0
c	**1**	1	0	**1**	0	1
d	**1**	0	1	**1**	1	0
e	**1**	1	0	**0**	0	0

The variable **a** would be chosen to spilt on, since it occurs most often in the smallest clauses (in this case binary clauses). In the case of a tie, it would seem appropriate to consider the next smallest clauses. although many implementations of the mom's heuristic consider only binary clauses and disregard all others (for optimisation).

[5]There may be more than one candidate, so a tie-breaking heuristic is sometimes used.

Some implementations of the mom's heuristic favour balanced variables (those which occur frequently both positively and negatively).

Variable	Occurrences in clauses of length 2	positive	negative
p	**4**	3	1
q	**4**	2	2
r	**9**	1	8
s	**7**	4	3

For instance **q** would be chosen over **p** as it is the more balanced variable. The motivation for this choice is that it maximises the effect of unit propagation in *both* subtrees arising from the branch. This principle can be extended to favour balanced variables over more frequently occurring unbalanced ones (e.g. **s** being preferable to **r**).

Let $n(l)$ be the number of times literal l occurs in the shortest clauses

The basic mom's heuristic chooses the variable x for which $n(x)+n(\neg x)$ is maximal, while the balanced mom's heuristic maximises $min(n(x), n(\neg x))$.

The generalised moms heuristic

Variable	Occurrences in clauses of length 2	Occurrences in clauses of length 3
x	**5**	**1**
y	**4**	**8**

In the above example **x** would be chosen by the moms heuristic, but should it be? It has been suggested that all clauses should be considered with larger clauses being given an exponentially decreasing weight [13], [12]. The Jeroslow-Wang branching rule chooses the literal l which maximises $J(l)$.

$$J(l) = \sum_{c \in C_l} 2^{n_c}$$

Where C_l is the set of all clauses which contain l, and n_c is the number of literals in clause c.

Chu Min Li [14] suggests (from the result of empirical tests) that a clause of length n should be counted as 5 clauses of length $n+1$ (in particular 1 binary clause is worth 5 ternary clauses). Thus in the example above, **y** would be chosen. This is only a small extension to the basic moms heuristic however and does not show significant performance gains over the basic moms heuristic, possibly because this situation occurs infrequently.

The unit propagation lookahead heuristic

The unit propagation (UP) heuristic is quite different from the moms heuristic, although it has the same aim — to maximise the effect of unit propagation in the subtrees resulting from the branch. The UP heuristic calculates the number of clauses that would be shortened (have fewer literals) from branching on each variable (the "lookahead") and chooses the variable which shortens the most clauses. This is achieved by examining a set of variables at each branch; a variable x in a formula f is examined by the following procedure:

- The formula p is obtained by performing unit propagation on $f \wedge x$.
- The formula q is obtained by performing unit propagation on $f \wedge \neg x$.
- The individual weights of x and $\neg x$ are given by $w(x) = diff(p, f)$ and $w(\neg x) = diff(q, f)$, where $diff(a, b)$ is the number of clauses that are shorter in a than in b.
- The overall weight[6]given to the variable is $H(x) = w(x) \times w(\neg x) \times 1024 + w(x) + w(\neg x)$.

At each node in the search, the pure UP heuristic examines all variables that appear in the formula, choosing the variable x that has the highest $H(x)$ value. To examine every variable is computationally expensive, so a restriction on the variables examined is a good compromise between the speed and accuracy of the heuristic, to obtain the overall fastest procedure. Chu Min Li [14] suggests that a restriction based on the familiar moms heuristic performs well, where a variable is examined if it has at least i binary occurrences, of which at least j are positive and j negative. Suggested values for i and j are $1 \leq i \leq 4$ and $0 \leq j \leq 2$. Intuitively this restricts the heuristic to variables which are most constrained and therefore more likely to be the best branching candidates.

It is widely appreciated that more variables should be examined earlier on in the tree, to increase the power of the heuristic in the more crucial early branches, and all implementations of the UP heuristic attempt to do this. POSIT (Freeman) has a maximum bound on the number of variables examined, as a function of the search tree depth. Satz (Li) shows performance gains from a *minimum* bound where all variables are examined if the number of variables determined by the restriction is less than this minimum. As an example, early on in the search there are usually few constrained variables so the restriction would not usually give enough variables to exceed the minimum bound, and all variables would be examined. This "dynamic" bound on the number of examined variables allows performance gains by adapting to the current state of the search — if there are few constrained variables (anywhere in the tree) then many variables will be examined, but if there are a sufficient number of constrained variables then only these will be examined.

A second benefit of the UP heuristic is that it can detect, for each variable examined, which values falsify the formula after the unit propagation step (commonly called failed literals). The UP heuristic can be easily adapted to detect failed literals simply by testing if each unit propagation lookahead step produces an empty clause (a trivial addition to the unit propagation step).

2.2.4 Value order

Each variable branch generates two subtrees. Assuming a depth-first implementation of DP, the subtrees must be explored in some order — this is the value order. There has been a lot of disagreement as to whether the value order has an affect on the performance of DP.

C-SAT [7] chooses the value that satisfies as many literals (and thus clauses) as possible, intuitively choosing the branch that is more likely to lead to a solution. Freeman [9] suggests that the value should be chosen that falsifies as many literals in short clauses as possible. Although this technique seems intuitively to choose the subtree less likely to contain a solution, it maximises the effect of unit propagation in the first subtree explored, thus reaching a global solution quickly if one exists. Both these strategies are based on sound intuitive arguments but they are in conflict with each other — you can't please

[6]Suggested by Freeman in [9], to balance the number of positive and negative occurrences.

both. Furthermore, Freeman compares his recommended method to a fixed value order (i.e. for branching variable $\mathbf{x}$, first explore $\mathbf{x} \mapsto T$ then $\mathbf{x} \mapsto F$) and shows that *there is no significant performance difference, either in the search tree size or run time* on hard random 3-SAT formulas. Tableau [4] uses a fixed value order, as does satz, which is currently one of the best performing DP solvers. It is worth mentioning that the value order does not affect the performance of the DP procedure on unsatisfiable formulas since the whole tree must be explored.

My opinion on the topic is that there is no clear winner out of all the value ordering strategies presented. Theoretically speaking, any satisfiable formula has it's solution (or one of them) expressible as a function of the original formula. Given a suitably powerful technique (for instance the function from all possible satisfiable formulas to their respective satisfying assignments), satisfiability can be determined in a single step simply by examining the formula. Taking this argument to the value ordering level, at any branch the best subtree is a function of the formula. Although I do not argue that the value order does not affect the search, no value order which I am aware of is significantly better on a range of problems than a fixed one.

2.2.5 Other modifications to the DP procedure

Preprocessing techniques

POSIT uses a technique called initial k-literal deletion, described in the original Davis-Putnam article [5]. This technique eliminates variables from the formula that have k positive or k negative occurrences.

If the literal x occurs in the clauses $(x \vee C_1), (x \vee C_2), \ldots, (x \vee C_k)$ and it's complement $\overline{x}$ occurs in the clauses $(\overline{x} \vee C'_1), (\overline{x} \vee C'_2), \ldots, (\overline{x} \vee C'_{k'})$, x is eliminated by deleting these clauses and replacing them with $\bigwedge_{i=1}^{k} \bigwedge_{j=1}^{k'} (C_i \vee C'_j)$.

This typically increases both the number and size of the clauses in the formula . It is too costly to perform at each node in the tree and only has limited use at the root — Freeman concludes that search performance is only improved for $k = 1$ and $k = 2$. Many formulas do not have many 1- and 2-literals and therefore cannot be simplified in this way.

A limited form of resolution can sometimes be used to refine the input formula before the search begins (again, it is usually too slow to use during the search). It can only be used on certain classes of problems.

Problem-dependent techniques

If more information about the problem is available, various tweaks can be made to the heuristic to increase performance. For randomly generated formulas, a version of tableau takes as it's input the formula and which parameters where used to generate it. The behaviour of the heuristic is then modified given this information.

Symmetry detection is the technique of detecting pairs of variables which can be interchanged without affecting the overall structure of the formula. This is obviously limited to certain classes of structured problems, but can improve performance significantly on these types of problems, removing redundancy from the search.

Randomisation and restarts

By incorporating randomisation in the heuristic, the Davis-Putnam procedure can be made non-deterministic. The search can then be given a time bound and re-started if it exceeds this. The time bound is initially small, but increases during the search. This technique has generated good results, on several classes of problems [9].

Integration with local search

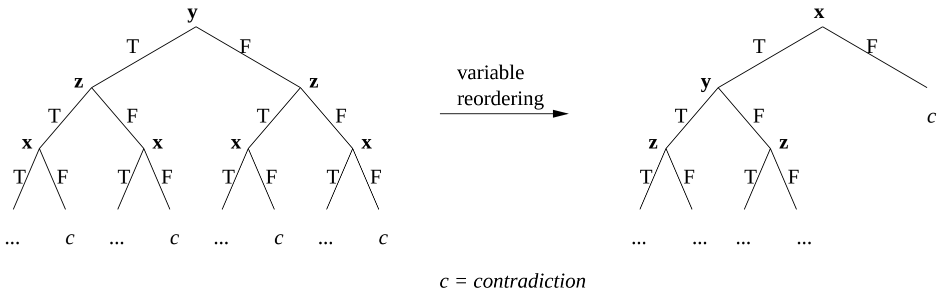
Local search procedures can be used in conjunction with the Davis-Putnam procedure. The input formula is run on local search procedure; if it does not find a solution, the final assignment is input to DP, which uses it to tailor the heuristic. This has been shown by James Crawford in [3] to improve performance by about 10%. Crawford tests on random formulas but hypothesises that this technique would better on structured problems.

Parallelism

The Davis-Putnam procedure is well-suited to coarse-grained parallelism, since the tree can be divided up, and each sub-tree can be explored independently. It is, however, a brute-force method, and doubling the number of processors working on the problem will only permit a constant increase in the feasible problem size. A number of different techniques can be used in parallel (e.g. DP, local search, genetic algorithms) to offset each other's disadvantages.

Look-back techniques

The backtracking search can often explore regions of the search space which contain no solutions or rediscover the same contradictions repeatedly. For example, if the assignment $\mathbf{x} \mapsto F$ leads to a contradiction, the following search could be carried out, repeatedly doing the same work. This incorrect variable ordering can be cured to some extent by using a strong heuristic (e.g. the UP heuristic detects failed literals).



When a contradiction is discovered, chronological backtracking explores the most recent unexplored subtree. This may be a branch which was not responsible for the contradiction, so the same contradiction will be found again. There are two main approaches to tackling this problem: selective backtracking and lateral pruning. Selective backtracking consists of testing whether each potential backtracking point was responsible for the failure and skipping over it if it was not. Lateral pruning determines a new constraint each time it finds a contradiction and adds it to the current problem, to ensure that the same contradiction is not repeatedly found. A search procedure which tries to avoid these problems is said to have dependency-directed backtracking. If implemented efficiently, it is particularly useful on certain classes of structured problems.

2.3 Non-Boolean formulas

Some problems are not suited to a Boolean encoding, but are more naturally expressed in terms of variables with domain sizes greater than two. Non-Boolean formulas are logical formulas, just as Boolean formulas are, but they contain variables with an arbitrary (but finite) domain size. They are introduced in [10].

A non-Boolean literal consists of a variable and an associated value set (consisting of values taken from the variable's domain). For a non-Boolean literal with variable $\mathbf{X}$ with value set $\{a_1,a_2,a_3,\ldots\}$, the semantics are defined as:

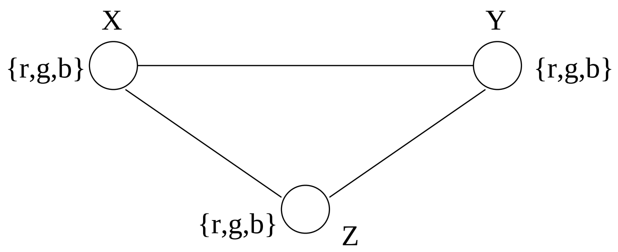
$\mathbf{X}\in\{a_1,a_2,a_3,\ldots\}$	True under an assignment which contains $\mathbf{X} \mapsto a_1$ or $\mathbf{X} \mapsto a_2$ or $\mathbf{X} \mapsto a_3$, …; False otherwise.

It can be seen that non-Boolean literals evaluate to either True or False, as in the Boolean case.

The logical operators $\neg$, $\vee$ and $\wedge$ used in non-Boolean formulas have the same semantics as those in Boolean formulas. For any non-Boolean variable $\mathbf{X}$ with domain D and any non-empty value set $\{a_1,a_2,\ldots,a_n\} \subseteq D$, $\mathbf{X}\in\{a_1,a_2,\ldots,a_n\}$ is a non-Boolean formula. For any non-Boolean formulas $\mathbf{A}$ and $\mathbf{B}$, $\neg\mathbf{A}$, $\mathbf{A} \vee \mathbf{B}$ and $\mathbf{A} \wedge \mathbf{B}$ are non-Boolean formulas with semantics identical to the Boolean case.

Examples of problems whose encoding into a non-Boolean (NB) formula is simpler than a Boolean encoding are the n-queens problem, constraint satisfaction problems, the Hamiltonian circuit problem, graph colouring and planning.

A graph colouring problem consists of an undirected graph and two constraints: that each node has exactly one colour (taken from a set of available colours) and that no two adjacent nodes may be the same colour. The task is to decide if there exists a colouring that satisfies these constraints and if so determine what it is. The figure below shows a simple graph, along with the possible colouring domains for each node. This simple example illustrates the contrast between the direct non-Boolean encoding and the Boolean translation.



The problem is intuitively expressed as:

$f = \mathbf{X} \neq \mathbf{Y} \;\wedge$	nodes X and Y must not both be the same colour
$\mathbf{X} \neq \mathbf{Z} \;\wedge$	nodes X and Z must not both be the same colour
$\mathbf{Y} \neq \mathbf{Z}$	nodes Y and Z must not both be the same colour

This can be converted into a non-Boolean formula according to the previously described syntax.

$$\begin{array}{rlllll} f_{NB} = & \neg(\mathbf{X}{\in}\{\mathrm{r}\} \wedge \mathbf{Y}{\in}\{\mathrm{r}\}) & \wedge & \neg(\mathbf{X}{\in}\{\mathrm{g}\} \wedge \mathbf{Y}{\in}\{\mathrm{g}\}) & \wedge & \neg(\mathbf{X}{\in}\{\mathrm{b}\} \wedge \mathbf{Y}{\in}\{\mathrm{b}\}) \quad \wedge \\ & \neg(\mathbf{X}{\in}\{\mathrm{r}\} \wedge \mathbf{Z}{\in}\{\mathrm{r}\}) & \wedge & \neg(\mathbf{X}{\in}\{\mathrm{g}\} \wedge \mathbf{Z}{\in}\{\mathrm{g}\}) & \wedge & \neg(\mathbf{X}{\in}\{\mathrm{b}\} \wedge \mathbf{Z}{\in}\{\mathrm{b}\}) \quad \wedge \\ & \neg(\mathbf{Y}{\in}\{\mathrm{r}\} \wedge \mathbf{Z}{\in}\{\mathrm{r}\}) & \wedge & \neg(\mathbf{Y}{\in}\{\mathrm{g}\} \wedge \mathbf{Z}{\in}\{\mathrm{g}\}) & \wedge & \neg(\mathbf{Y}{\in}\{\mathrm{b}\} \wedge \mathbf{Z}{\in}\{\mathrm{b}\}) \end{array}$$

Applying DeMorgan's law allows the formula to be re-written in CNF.

$$\begin{array}{rlllll} f_{NB} = & (\neg\mathbf{X}{\in}\{\mathrm{r}\} \vee \neg\mathbf{Y}{\in}\{\mathrm{r}\}) & \wedge & (\neg\mathbf{X}{\in}\{\mathrm{g}\} \vee \neg\mathbf{Y}{\in}\{\mathrm{g}\}) & \wedge & (\neg\mathbf{X}{\in}\{\mathrm{b}\} \vee \neg\mathbf{Y}{\in}\{\mathrm{b}\}) \quad \wedge \\ & (\neg\mathbf{X}{\in}\{\mathrm{r}\} \vee \neg\mathbf{Z}{\in}\{\mathrm{r}\}) & \wedge & (\neg\mathbf{X}{\in}\{\mathrm{g}\} \vee \neg\mathbf{Z}{\in}\{\mathrm{g}\}) & \wedge & (\neg\mathbf{X}{\in}\{\mathrm{b}\} \vee \neg\mathbf{Z}{\in}\{\mathrm{b}\}) \quad \wedge \\ & (\neg\mathbf{Y}{\in}\{\mathrm{r}\} \vee \neg\mathbf{Z}{\in}\{\mathrm{r}\}) & \wedge & (\neg\mathbf{Y}{\in}\{\mathrm{g}\} \vee \neg\mathbf{Z}{\in}\{\mathrm{g}\}) & \wedge & (\neg\mathbf{Y}{\in}\{\mathrm{b}\} \vee \neg\mathbf{Z}{\in}\{\mathrm{b}\}) \end{array}$$

2.4 Translating non-Boolean formulas into Boolean formulas

All the translation methods presented here are previously described in [10].

2.4.1 The unary translation

The straightforward Boolean translation (the *unary* translation) of this formula is achieved by creating a Boolean variable for every (NB-variable, value) pair. For each NB-variable **X** and value **a** from the domain, a Boolean variable **x_a** is created; the interpretation of the assignment is that $\mathbf{X} \mapsto \mathbf{a}$ if $\mathbf{x\_a} \mapsto T$. For instance **z_r**, **z_g** and **z_b** would be created to stand for the three possible values that the NB-variable **Z** may take in the previous example.

Each literal $\mathbf{X}{\in}\{\mathrm{a}_1,\mathrm{a}_2,\ldots,\mathrm{a}_n\}$ is translated as $\mathbf{x\_a}_1 \vee \mathbf{x\_a}_2 \vee \ldots \vee \mathbf{x\_a}_n$. The resulting Boolean formula is known as the kernel of the translation.

$$\begin{array}{rlllll} \mathit{kernel} = & (\neg\mathbf{x\_r} \vee \neg\mathbf{y\_r}) & \wedge & (\neg\mathbf{x\_g} \vee \neg\mathbf{y\_g}) & \wedge & (\neg\mathbf{x\_b} \vee \neg\mathbf{y\_b}) \quad \wedge \\ & (\neg\mathbf{x\_r} \vee \neg\mathbf{z\_r}) & \wedge & (\neg\mathbf{x\_g} \vee \neg\mathbf{z\_g}) & \wedge & (\neg\mathbf{x\_b} \vee \neg\mathbf{z\_b}) \quad \wedge \\ & (\neg\mathbf{y\_r} \vee \neg\mathbf{z\_r}) & \wedge & (\neg\mathbf{y\_g} \vee \neg\mathbf{z\_g}) & \wedge & (\neg\mathbf{y\_b} \vee \neg\mathbf{z\_b}) \end{array}$$

The space of possible variable assignments (the search space) is now larger than that of the corresponding NB-formula ($2^9 = 512$ possible assignments compared to $3^3 = 27$). To preserve the singular nature of the non-Boolean encoding, extra clauses must be added to the translation. A non-singular variable assignment of the Boolean translation is where a corresponding NB-variable is not given a single value. For each NB-variable **X**, with domain size n, an assignment is singular if exactly one of the corresponding Boolean variables **x_1**, **x_2**, $\ldots$, **x_n** is True. Examples of non-singular assignments from the above example are: $\{\mathbf{z\_r} \mapsto F, \mathbf{z\_g} \mapsto F, \mathbf{z\_b} \mapsto F\}$ and $\{\mathbf{z\_r} \mapsto T, \mathbf{z\_g} \mapsto T, \mathbf{z\_b} \mapsto F\}$. Non-singular assignments are enforced by the inclusion of extra clauses (the at least one (ALO) and at most one (AMO) formulas).

The ALO clauses stipulate that each corresponding NB-variable must be assigned at least one value. For each NB-variable **X** with domain size n, the clause $(\mathbf{x\_1} \vee \mathbf{x\_2} \vee \ldots \vee \mathbf{x\_n})$ is present in the ALO sub-formula.

$$
\begin{aligned}
ALO = \quad & (\mathbf{x\_r} \vee \mathbf{x\_g} \vee \mathbf{x\_b}) \quad \wedge \\
& (\mathbf{y\_r} \vee \mathbf{y\_g} \vee \mathbf{y\_b}) \quad \wedge \\
& (\mathbf{z\_r} \vee \mathbf{z\_g} \vee \mathbf{z\_b})
\end{aligned}
$$

The AMO clauses state that for each NB-variable must be assigned at most one value. For each NB-variable $\mathbf{X}$ and any values i and j from the domain, $\mathbf{x\_i}$ and $\mathbf{x\_j}$ must not both be True in the Boolean translation. Thus for all NB-variables $\mathbf{X}$ and values i and j, the clause $(\neg\mathbf{x\_i} \vee \neg\mathbf{x\_j})$ is included in the AMO clauses.

$$
\begin{aligned}
AMO = \quad & (\neg\mathbf{x\_g} \vee \neg\mathbf{x\_b}) \quad \wedge \quad (\neg\mathbf{x\_r} \vee \neg\mathbf{x\_b}) \quad \wedge \quad (\neg\mathbf{x\_r} \vee \neg\mathbf{x\_g}) \quad \wedge \\
& (\neg\mathbf{y\_g} \vee \neg\mathbf{y\_b}) \quad \wedge \quad (\neg\mathbf{y\_r} \vee \neg\mathbf{y\_b}) \quad \wedge \quad (\neg\mathbf{y\_r} \vee \neg\mathbf{y\_g}) \quad \wedge \\
& (\neg\mathbf{z\_g} \vee \neg\mathbf{z\_b}) \quad \wedge \quad (\neg\mathbf{z\_r} \vee \neg\mathbf{z\_b}) \quad \wedge \quad (\neg\mathbf{z\_r} \vee \neg\mathbf{z\_g})
\end{aligned}
$$

The conjunction of the kernel, ALO clauses and AMO clauses gives the unary Boolean translation.

$$
\mathbf{Translation = kernel} \wedge \mathbf{ALO} \wedge \mathbf{AMO}
$$

$$
\begin{aligned}
f_B = \quad & (\neg\mathbf{x\_r} \vee \neg\mathbf{y\_r}) \quad \wedge \quad (\neg\mathbf{x\_g} \vee \neg\mathbf{y\_g}) \quad \wedge \quad (\neg\mathbf{x\_b} \vee \neg\mathbf{y\_b}) \quad \wedge \\
& (\neg\mathbf{x\_r} \vee \neg\mathbf{z\_r}) \quad \wedge \quad (\neg\mathbf{x\_g} \vee \neg\mathbf{z\_g}) \quad \wedge \quad (\neg\mathbf{x\_b} \vee \neg\mathbf{z\_b}) \quad \wedge \\
& (\neg\mathbf{y\_r} \vee \neg\mathbf{z\_r}) \quad \wedge \quad (\neg\mathbf{y\_g} \vee \neg\mathbf{z\_g}) \quad \wedge \quad (\neg\mathbf{y\_b} \vee \neg\mathbf{z\_b}) \quad \wedge \\
& (\mathbf{x\_r} \vee \mathbf{x\_g} \vee \mathbf{x\_b}) \quad \wedge \quad (\mathbf{y\_r} \vee \mathbf{y\_g} \vee \mathbf{y\_b}) \quad \wedge \quad (\mathbf{z\_r} \vee \mathbf{z\_g} \vee \mathbf{z\_b}) \quad \wedge \\
& (\neg\mathbf{x\_g} \vee \neg\mathbf{x\_b}) \quad \wedge \quad (\neg\mathbf{x\_r} \vee \neg\mathbf{x\_b}) \quad \wedge \quad (\neg\mathbf{x\_r} \vee \neg\mathbf{x\_g}) \quad \wedge \\
& (\neg\mathbf{y\_g} \vee \neg\mathbf{y\_b}) \quad \wedge \quad (\neg\mathbf{y\_r} \vee \neg\mathbf{y\_b}) \quad \wedge \quad (\neg\mathbf{y\_r} \vee \neg\mathbf{y\_g}) \quad \wedge \\
& (\neg\mathbf{z\_g} \vee \neg\mathbf{z\_b}) \quad \wedge \quad (\neg\mathbf{z\_r} \vee \neg\mathbf{z\_b}) \quad \wedge \quad (\neg\mathbf{z\_r} \vee \neg\mathbf{z\_g})
\end{aligned}
$$

It turns out that the AMO clauses can be omitted for graph colouring problems, since a solution that colours a node with more than one colour can be transformed into one that colours the node with a single colour, by simply choosing any one of the assigned colours (all clauses relating to that node will still be satisfied). This is intuitively understood because the kernel consists solely of negative literals — changing a variable from True to False will not cause any clauses to be falsified.

Non-singular assignments can be split into two groups: those which are *empty* and those which are *multiple*. Some examples are shown below, on the NB-variable $\mathbf{X}$, over the domain {a,b,c,d}.

x_a	**x_b**	**x_c**	**x_d**	Non-Boolean interpretation	Singular/non-singular
T	*F*	*F*	*F*	$\mathbf{X} \mapsto$ a	singular
F	*F*	***T***	*F*	$\mathbf{X} \mapsto$ c	singular
F	*F*	*F*	*F*	$\mathbf{X} \mapsto$ nothing	non-singular (empty)
F	***T***	*F*	***T***	$\mathbf{X} \mapsto$ b or d	non-singular (multiple)

Positive and negative formulas

A (Boolean or non-Boolean) formula is *positive* if it contains only positive literals, it is *negative* if it contains only negative literals. By changing a variable from False to True in a positive Boolean CNF formula, all clauses which were previously satisfied will remain satisfied (i.e. more clauses may be satisfied but none will be broken). The opposite is true for negative Boolean CNF formulas, where changing a variable from True to False preserves satisfied clauses.

Formula					Positive/Negative?
$(\mathbf{x} \vee \mathbf{y})$	$\wedge$	$(\mathbf{x} \vee \mathbf{z})$	$\wedge$	$(\mathbf{y} \vee \mathbf{z})$	Positive
$(\neg\mathbf{p} \vee \neg\mathbf{q})$	$\wedge$	$(\neg\mathbf{p} \vee \neg\mathbf{r})$	$\wedge$	$(\neg\mathbf{q} \vee \neg\mathbf{r})$	Negative
$(\mathbf{a} \vee \neg\mathbf{b})$	$\wedge$	$(\neg\mathbf{b} \vee \neg\mathbf{c})$	$\wedge$	$(\neg\mathbf{a} \vee \mathbf{c})$	—

For any unary translation, if the kernel is positive then the ALO is unnecessary, since any variable can be changed from False to True without breaking any kernel clauses. If we exclude the ALO clauses from the translation (but, of course, include the AMO — to enforce non-multiple assignments) any solution which contains empty assignments can be transformed into a solution which contains only singular assignments. To do this each NB-variable **X** which is assigned no value ($\mathbf{x\_1} \mapsto F$, $\mathbf{x\_2} \mapsto F$, $\ldots$, $\mathbf{x\_n} \mapsto F$) pick one variable from $\mathbf{x\_1}$, $\mathbf{x\_2}$, $\ldots$, $\mathbf{x\_n}$ and assign it to True. This cannot break any clauses, either in the kernel or the AMO. For translations with negative kernels the opposite is True, where the AMO is unnecessary (but the ALO must still be included). In general, a positive NB formula gives rise to a positive kernel under the unary translation and a negative NB formula produces a negative kernel.

The effect of the ALO and AMO clauses on the Davis-Putnam procedure

As previously mentioned, the size of the search space is increased (sometimes, quite dramatically) in going from the NB-formula to the unary translation. The extra clauses added to enforce singular assignments prevent the Boolean search from actually exploring these non-singular assignments, so although the potential search space is enlarged, none of it will actually be *explored*[7] (in contrast to local search methods such as those in [10]). This is quite an important point, as the non-Boolean search procedure does not have as much of an advantage over the translation method as it might first appear.

The downside of translating and solving is that the extra clauses hamper the search by storing the singular restriction in the data structures, rather than implicitly in the algorithm. The effect of this is that the data structures must be modified as the search progresses, which takes time. Obviously this effect is greater for formulas with more ALO and AMO clauses.

An additional effect is that the extra clauses in the translation (ALO and AMO) may make the heuristic less powerful. The moms heuristic, which picks the variable which occurs most often in the smallest clauses, would give much attention to the AMO clauses (all binary). In the AMO, all unassigned variables appear with relatively equal frequencies, so the kernel formula will have less effect on which variable is chosen. The pure UP heuristic will not be affected by the extra clauses in this way, although the lookahead step may be slower because of the extra clauses (the above point). It is not clear how the UP heuristic with the moms restriction (as in satz) will perform.

2.4.2 The binary translation

Another method for translating non-Boolean formulas into equivalent Boolean formulas is the *binary* translation. In the unary translation one Boolean variable represented a single value for an NB-variable, in the binary translation each Boolean variable represents two values. For the NB-variable **X** and

[7] The search may deviate into a state corresponding to a non-singular (partial) assignment, but it will immediately backtrack because one of the ALO or AMO clauses will be violated.

domain {1,2,3,4}, the Boolean encoding would use two variables, **x_a** and **x_b**, with the non-Boolean interpretation being defined as:

x_a	**x_b**	Non-Boolean interpretation
F	*F*	$\mathbf{X} \mapsto 1$
F	***T***	$\mathbf{X} \mapsto 2$
T	*F*	$\mathbf{X} \mapsto 3$
T	***T***	$\mathbf{X} \mapsto 4$

In general, for domain size d this method uses $\lceil \log_2 d \rceil$ Boolean variables for each non-Boolean variable (the unary encoding uses d Boolean variables). This does not allow non-singular assignments in the search space, but unfortunately can cause large growth in the formula when re-written in CNF — the innermost conjunctions may need to be distributed outwards[8]. This can make the binary translation unfeasibly large for many formulas. An example of this translation method is given below, the conversion to CNF has been omitted.

$$
\begin{array}{lccccc}
\text{non-Boolean clause} & \mathbf{X}{\in}\{1{,}2\} & \vee & \mathbf{Y}{\in}\{1{,}3\} & \vee & \mathbf{Z}{\in}\{2{,}4\} \\[1em]
\text{binary translation} & (\neg\mathbf{x\_a} \wedge \neg\mathbf{x\_b}) & \vee & (\neg\mathbf{y\_a} \wedge \neg\mathbf{y\_b}) & \vee & (\neg\mathbf{z\_a} \wedge \mathbf{z\_b}) \quad \vee \\
 & (\neg\mathbf{x\_a} \wedge \mathbf{x\_b}) & \vee & (\mathbf{y\_a} \wedge \neg\mathbf{y\_b}) & \vee & (\mathbf{z\_a} \wedge \mathbf{z\_b})
\end{array}
$$

2.4.3 The unary-binary translation

The unary-binary translation uses the same unary encoding for the kernel formula, but the clauses to enforce singular assignment have the binary encoding. Therefore the search space is the same, but again, not all of it will be explored. Extra variables are used in the clauses to enforce singular assignments ($\lceil \log_2 d \rceil$ for each variable), but the number of clauses is often less, giving smaller formulas. For a non-Boolean variable **X** with domain {1,2,3}, singular assignments are enforced by ensuring for all values of **x_a** and **x_b**, exactly one of **x_1**, **x_2** and **x_3** are true. In this example, the domain size is not a power of two so the remaining combinations **x_a** and **x_b** must be constrained not to occur.

$$
\begin{array}{ll}
 & (\mathbf{x\_1} \Leftrightarrow \neg\mathbf{x\_a} \wedge \neg\mathbf{x\_b}) \quad \wedge \\
 & (\mathbf{x\_2} \Leftrightarrow \neg\mathbf{x\_a} \wedge \mathbf{x\_b}) \quad \wedge \\
 & (\mathbf{x\_3} \Leftrightarrow \mathbf{x\_a} \wedge \neg\mathbf{x\_b}) \quad \wedge \\
 & \neg(\mathbf{x\_a} \wedge \mathbf{x\_b}) \\[1em]
\equiv & (\mathbf{x\_1} \Rightarrow \neg\mathbf{x\_a} \wedge \neg\mathbf{x\_b}) \quad \wedge \quad (\neg\mathbf{x\_a} \wedge \neg\mathbf{x\_b} \Rightarrow \mathbf{x\_1}) \quad \wedge \\
 & (\mathbf{x\_2} \Rightarrow \neg\mathbf{x\_a} \wedge \mathbf{x\_b}) \quad \wedge \quad (\neg\mathbf{x\_a} \wedge \mathbf{x\_b} \Rightarrow \mathbf{x\_2}) \quad \wedge \\
 & (\mathbf{x\_3} \Rightarrow \mathbf{x\_a} \wedge \neg\mathbf{x\_b}) \quad \wedge \quad (\mathbf{x\_a} \wedge \neg\mathbf{x\_b} \Rightarrow \mathbf{x\_3}) \quad \wedge \\
 & \neg(\mathbf{x\_a} \wedge \mathbf{x\_b}) \\[1em]
\equiv & (\neg\mathbf{x\_1} \vee \neg\mathbf{x\_a}) \wedge (\neg\mathbf{x\_1} \vee \neg\mathbf{x\_b}) \quad \wedge \quad (\mathbf{x\_a} \vee \mathbf{x\_b} \vee \mathbf{x\_1}) \quad \wedge \\
 & (\neg\mathbf{x\_2} \vee \neg\mathbf{x\_a}) \wedge (\neg\mathbf{x\_2} \vee \mathbf{x\_b}) \quad \wedge \quad (\mathbf{x\_a} \vee \neg\mathbf{x\_b} \vee \mathbf{x\_2}) \quad \wedge \\
 & (\neg\mathbf{x\_3} \vee \mathbf{x\_a}) \wedge (\neg\mathbf{x\_3} \vee \neg\mathbf{x\_b}) \quad \wedge \quad (\neg\mathbf{x\_a} \vee \mathbf{x\_b} \vee \mathbf{x\_3}) \quad \wedge \\
 & (\neg\mathbf{x\_a} \vee \neg\mathbf{x\_b})
\end{array}
$$

[8]For negative non-Boolean formulas, this is not a problem since $\vee$ is still the innermost operator.

2.4.4 Translating negative literals

Any non-Boolean literal can be represented both positively and negatively. For non-Boolean variable $\mathbf{X}$ with domain D, the following are equivalent.

$$\neg\mathbf{X}\in\{a_1,a_2,a_3,\ldots\} \quad\equiv\quad \mathbf{X}\notin\{a_1,a_2,a_3,\ldots\} \quad\equiv\quad \mathbf{X}\in(D\backslash\{a_1,a_2,a_3,\ldots\})$$

For certain NB formulas (for example graph colouring), the formula is more concisely written using negative literals instead of positive ones. For domain $\{a_1,a_2,a_3,\ldots,a_n\}$, the following are equivalent (simply a special case of the above equivalence).

$$\mathbf{X}\in\{a_1,a_2,\ldots,a_{k-1},a_{k+1},\ldots,a_n\} \quad\equiv\quad \neg\mathbf{X}\in\{a_k\}$$

For formulas of this kind, using the negative encoding on the NB formula produces much smaller Boolean translations, especially for large domains. This increases the speed of the Boolean search on the translation.

Negative literals such as $\neg\mathbf{X}\in\{a_j,a_k\}$ can be smaller than their positive equivalents, but translating a formula full of these will result in many clauses when expressed in CNF. For domain {a,b,c,d,e,f}, the clause $\neg\mathbf{X}\in\{a,d\} \vee \neg\mathbf{Y}\in\{b,e\} \vee \neg\mathbf{Z}\in\{c,f\}$ is translated as:

$$\neg\mathbf{X}\in\{a,d\} \vee \neg\mathbf{Y}\in\{b,e\} \vee \neg\mathbf{Z}\in\{c,f\}$$

$$\neg(\mathbf{x\_a} \vee \mathbf{x\_d}) \;\vee\; \neg(\mathbf{y\_b} \vee \mathbf{y\_e}) \;\vee\; \neg(\mathbf{z\_c} \vee \mathbf{z\_f})$$

$$\equiv\; (\neg\mathbf{x\_a} \wedge \neg\mathbf{x\_d}) \;\vee\; (\neg\mathbf{y\_b} \wedge \neg\mathbf{y\_e}) \;\vee\; (\neg\mathbf{z\_c} \wedge \neg\mathbf{z\_f})$$

$$\begin{aligned}\equiv\; & (\neg\mathbf{x\_a} \vee \neg\mathbf{y\_b} \vee \neg\mathbf{z\_c}) \;\wedge\; (\neg\mathbf{x\_d} \vee \neg\mathbf{y\_b} \vee \neg\mathbf{z\_c}) \;\wedge \\ & (\neg\mathbf{x\_a} \vee \neg\mathbf{y\_e} \vee \neg\mathbf{z\_c}) \;\wedge\; (\neg\mathbf{x\_d} \vee \neg\mathbf{y\_e} \vee \neg\mathbf{z\_c}) \;\wedge \\ & (\neg\mathbf{x\_a} \vee \neg\mathbf{y\_b} \vee \neg\mathbf{z\_f}) \;\wedge\; (\neg\mathbf{x\_d} \vee \neg\mathbf{y\_b} \vee \neg\mathbf{z\_f}) \;\wedge \\ & (\neg\mathbf{x\_a} \vee \neg\mathbf{y\_e} \vee \neg\mathbf{z\_f}) \;\wedge\; (\neg\mathbf{x\_d} \vee \neg\mathbf{y\_e} \vee \neg\mathbf{z\_f})\end{aligned}$$

The positive equivalent, $\mathbf{X}\in\{b,c,e,f\} \vee \mathbf{Y}\in\{a,c,d,f\} \vee \mathbf{Z}\in\{a,b,d,e\}$, is translated as:

$$\mathbf{X}\in\{b,c,e,f\} \vee \mathbf{Y}\in\{a,c,d,f\} \vee \mathbf{Z}\in\{a,b,d,e\}$$

$$\mathbf{x\_b} \vee \mathbf{x\_c} \vee \mathbf{x\_e} \vee \mathbf{x\_f} \vee \mathbf{y\_a} \vee \mathbf{y\_c} \vee \mathbf{y\_d} \vee \mathbf{y\_f} \vee \mathbf{z\_a} \vee \mathbf{z\_b} \vee \mathbf{z\_d} \vee \mathbf{z\_e}$$

Therefore, it is not usually worth re-writing these types of formula as their negative equivalents before translating (to avoid the AMO clauses) because the translation is usually larger when it is expressed in CNF.

Design

3.1 Non-Boolean Davis-Putnam

Conceptually, the Boolean DP method is fairly simple:

When there is no choice to be made (when there is a unit clause or pure literal in the formula) then do the assignment and simplification, and recurse. Otherwise split up the problem (branch) committing to a new assignment in each subtree.

This general concept still holds in the non-Boolean case and provides a suitable level of abstraction to consider the NB-DP procedure.

Unit clauses may not consist of a variable with a single value, so unit propagation may now not fully assign a variable. The NB unit propagation step consists of the partial assignment of the variable (the removal of values which would contradict the unit clause variable) and the simplification of the resulting formula. The following example illustrates this.

$$\mathrm{f}_0 = \quad (\mathbf{X}{\in}\{\mathrm{a,b}\}) \quad \wedge \quad (\mathbf{X}{\in}\{\mathrm{a,b,c}\} \vee \alpha) \quad \wedge \quad (\mathbf{X}{\in}\{\mathrm{a,c}\} \vee \beta) \quad \wedge \quad (\mathbf{X}{\in}\{\mathrm{c,d}\} \vee \gamma)$$

$$\mathbf{X} \mapsto \{\mathrm{a,b}\}$$

The assignment of $\mathbf{X} \mapsto \{\mathrm{a,b}\}$ is read as "**X** is partially assigned to either a or b", and all other values are ruled out for **X** at this point in the search. After removing all other values from all literals **X** in the formula, all occurrences of $\mathbf{X}{\in}\{\mathrm{a,b}\}$ can be assigned True because they will be True whichever of {a,b} actually end up being assigned to **X** (they exactly match the constraint being propagated). Also, all occurrences of $\mathbf{X}{\in}\{\}$ can be assigned False, as they will never be satisfied by either $\mathbf{X} \mapsto \mathrm{a}$ or $\mathbf{X} \mapsto \mathrm{b}$. All other literals cannot be simplified to a truth value because they represent a tighter constraint than the original unit clause (e.g. $\mathbf{X}{\in}\{\mathrm{a}\}$ is more tightly constrained than $\mathbf{X}{\in}\{\mathrm{a,b}\}$).

$$\begin{array}{lccccccc}
\mathrm{f}_1 = & (\mathbf{X}{\in}\{\mathrm{a,b}\}) & \wedge & (\mathbf{X}{\in}\{\mathrm{a,b}\} \vee \alpha) & \wedge & (\mathbf{X}{\in}\{\mathrm{a}\} \vee \beta) & \wedge & (\mathbf{X}{\in}\{\} \vee \gamma) \\
\mathrm{f}_1 = & T & \wedge & (T \vee \alpha) & \wedge & (\mathbf{X}{\in}\{\mathrm{a}\} \vee \beta) & \wedge & (F \vee \gamma) \\
\mathrm{f}_1 = & & & & & (\mathbf{X}{\in}\{\mathrm{a}\} \vee \beta) & \wedge & \gamma
\end{array}$$

The pure literal rule is a little more general[9], including variables which occur in the formula with more than one value.

[9] This may seem trivial, but the Boolean implementation assumes that x is only pure if either x or $\neg x$ does not appear in the formula.

$$(\mathbf{X}{\in}\{a,b\} \vee \mathbf{Y}{\in}\{a,b\}) \quad \wedge \quad (\mathbf{X}{\in}\{a,c\} \vee \mathbf{Y}{\in}\{b,c\}) \quad \wedge \quad (\mathbf{X}{\in}\{a,d\}\ \mathbf{Y}{\in}\{c,d\})$$

Here, the value ‘a’ is pure since all occurrences of **X** appear with the value ‘a’. Note that this is the case even though ‘a’ is not the only value to appear with **X** in the formula (e.g. **X**∈{a,b} appears in the formula).

3.1.1 Non-Boolean branching

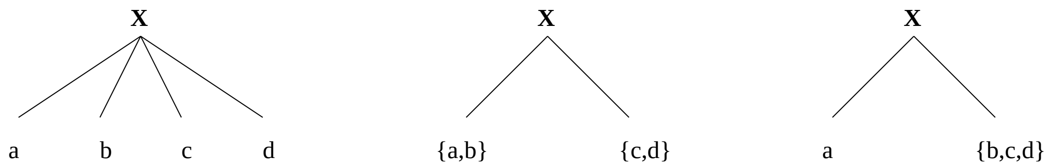
In the Boolean search, each branch is binary (it produces two subtrees). Additionally, each of the subtrees represents a full commitment to a particular value. These are both good properties.

- A small branching factor is more preferable than a large one because each branch produces fewer subtrees — for the most favourable branch[10], the worst subtree (the one most difficult to search) is more likely to be better if there are fewer subtrees. From the point of view of the heuristic, the optimal branch is more likely to give rise to more unit propagations in the worst of two subtrees than the worst of three.
- A full commitment to a value is useful in each subtree because it means that the most simplification can happen (unit propagation and pure literal elimination).

Those properties both occur in the Boolean search, but the non-Boolean search cannot have both, since branches either have to be non-binary or make only a partial commitment (to a subset of values). I have identified three approaches to non-Boolean branching.

1. For n possible values, each branch produces n subtrees, each making a full commitment to a single value.
2. The set of all values that could be explored is divided into two partitions. Each branch produces 2 subtrees, one making a partial commitment to one partition, the other making a partial commitment to the other partition.
3. For active values V, a branch on variable X and value a is binary, where one subtree makes the full commitment $X \mapsto a$ and the other makes the partial commitment $X \mapsto V \setminus a$.

The three branching strategies are illustrated below.



Strategies 2 and 3 introduce a further choice into the heuristic in addition to the choice of variable: a partitioning of the value set must be determined for the 2nd strategy and a single value must be chosen for the 3rd. The 3rd strategy is a less general case of the 2nd, but one subtree benefits from a full commitment to a value. In this strategy each branch is unbalanced.

The 2nd branching strategy is the most complex — to decide how to partition for optimal unit propagation is quite(!) difficult. For this reason, I decided against method 2, because I felt that the efficiency

[10] The branch with the best “worst subtree”.

of the search would depend heavily on the choice of partition at each branch. I did not have any ideas on how a good partitioning could be determined efficiently, since each value in a partition affects the unit propagation in it's subtree. The binary translation effectively operates like method, but does an arbitrary (but roughly equal) partitioning of the domain. Such a partitioning probably does not allow much simplification.

The 3rd strategy is a more specific instance of the 2nd, but commits to a single value in one subtree. The other is unlikely to produce much unit propagation since it commits to all other available values (i.e. all other values that have not been eliminated from inquiry). Sooner or later, this commitment must be refined by another branch, again removing another value. The branching variable should be chosen so as to have the maximum simplification effect on the formula (unit propagation). This process suffers from the same problem as the 2nd strategy — how are a set of values chosen to maximise unit propagation on the set? In this case, the number of possible partitions is reduced due to the restriction that one partition must consist of a single value, but the problem of optimising unit propagation on values *sets* still remains. This method is very similar to how the Boolean search operates on the unary translation, by exploring a single value at a time. It also has the same disadvantages, namely that there is little simplification that can be done in the partial-commitment subtree.

The 1st method does not suffer from this complexity, but more commitment is given than may be necessary. For example, if choosing $\mathbf{X} \mapsto$ a produces a similar subtree as choosing $\mathbf{X} \mapsto$ b, then redundant work can be avoided by choosing $\mathbf{X} \mapsto \{a,b\}$. This method may do more work than is necessary, but it's effectiveness does not depend on a difficult set partitioning decision. For this reason I chose to implement the 1st branching strategy.

3.1.2 Assumptions

I assume that all input formulas are positive, i.e. negative formulas are translated to equivalent positive formulas (see page 20). This is done to simplify the implementation and does not restrict the range of formulas that the non-Boolean search can operate on, although it may increase their size. Of course, when translating to a Boolean equivalent, the most efficient representation is used (i.e. graph colouring formulas are expressed as negative formulas, then translated).

I also assume that all clauses in the input formula contain at most one occurrence of each variable. If this is not the case, clauses can be re-written to obey this restriction.

$$\mathbf{X}{\in}\mathrm{A} \vee \mathbf{X}{\in}\mathrm{B} \vee \alpha \quad \equiv \quad \mathbf{X}{\in}(\mathrm{A}{\cup}\mathrm{B}) \vee \alpha$$

3.1.3 Abstract definition of non-Boolean unit propagation

I first present an abstract definition of non-Boolean unit propagation, which sets the scene for the more procedural explanation of the whole NB procedure. For a non-Boolean unit clause $\mathbf{X}{\in}\mathrm{A}$, all clauses $\mathbf{X}{\in}\mathrm{B} \vee \alpha$ are re-written according to the following rules:

1. If $\mathrm{A} \cap \mathrm{B} = \{\}$ then replace $\mathbf{X}{\in}\mathrm{B} \vee \alpha$ with α.
2. If $\mathrm{A} \subseteq \mathrm{B}$ then remove $\mathbf{X}{\in}\mathrm{B} \vee \alpha$ form the formula.
3. If $\mathrm{A} \supset \mathrm{B}$ then leave $\mathbf{X}{\in}\mathrm{B} \vee \alpha$ as it is.
4. Otherwise replace $\mathbf{X}{\in}\mathrm{B} \vee \alpha$ with $\mathbf{X}{\in}(\mathrm{A}{\cap}\mathrm{B}) \vee \alpha$.

3.1.4 Definition of the non-Boolean Davis-Putnam method

The Boolean method described on page 7 can be expanded to give the outline of the non-Boolean procedure. It is assumed that all variables have value domains D[11].

```
function NB_Davis-Putnam(formula f) return Boolean is
  /* unit propagation */
  repeat
    for each unit clause l in f do
      let l = X ∈ {a_1, a_2, ..., a_n}
      for all values a ∈ D \ {a_1, a_2, ..., a_n}
        replace all literals X ∈ {a, b_1, b_2, ..., b_n} that appear in f with X ∈ {b_1, b_2, ..., b_n}
      end for
      remove from f all clauses that contain the literal X ∈ {a_1, a_2, ..., a_n}
      remove X ∈ {} from all clauses in f in which it appears
    end for
    if f is empty then
      return True
    else if f contains the empty clause then
      return False
    end if
  until there are no unit clauses in f

  /* pure literal elimination */
  for all variables X whose occurrences in f all have the value a in their value set do
    remove from f all clauses that contain X
  end for

  /* branching */
  pick a variable X from f
  for all values a that X appears with in f do
    if NB_Davis-Putnam(f ∧ X ∈ {a}) then
      return True
    end if
  end for
  return False
end function
```

Again, the current assignment is kept track of, so the procedure can return a satisfying assignment if the formula is found to be satisfiable.

[11] This may not be true in the general case, but this assumption allows optimisations to be made in the implementation. Extra clauses can be used to constrain the domain of variables if necessary, for variable-specific domains.

3.1.5 An example of non-Boolean Davis-Putnam

As an example we use the previous graph colouring problem, re-written using only positive literals, for simplicity.

$$\begin{array}{rcccccc} f_0 = & (\mathbf{X}{\in}\{g,b\} \vee \mathbf{Y}{\in}\{g,b\}) & \wedge & (\mathbf{X}{\in}\{r,b\} \vee \mathbf{Y}{\in}\{r,b\}) & \wedge & (\mathbf{X}{\in}\{r,g\} \vee \mathbf{Y}{\in}\{r,g\}) & \wedge \\ & (\mathbf{X}{\in}\{g,b\} \vee \mathbf{Z}{\in}\{g,b\}) & \wedge & (\mathbf{X}{\in}\{r,b\} \vee \mathbf{Z}{\in}\{r,b\}) & \wedge & (\mathbf{X}{\in}\{r,g\} \vee \mathbf{Z}{\in}\{r,g\}) & \wedge \\ & (\mathbf{Y}{\in}\{g,b\} \vee \mathbf{Z}{\in}\{g,b\}) & \wedge & (\mathbf{Y}{\in}\{r,b\} \vee \mathbf{Z}{\in}\{r,b\}) & \wedge & (\mathbf{Y}{\in}\{r,g\} \vee \mathbf{Z}{\in}\{r,g\}) & \end{array}$$

$$A_0 = \{\}$$

There are no unit clauses so unit propagation cannot take place. Suppose $\mathbf{X}$ is chosen to branch on. Then we have the choices $\mathbf{X} \mapsto$ r, $\mathbf{X} \mapsto$ g and $\mathbf{X} \mapsto$ b. Suppose $\mathbf{X} \mapsto$ r is explored first.

$\mathbf{X} \mapsto$ r

$$\begin{array}{rcccccc} f_1 = & (F \vee \mathbf{Y}{\in}\{g,b\}) & \wedge & (T \vee \mathbf{Y}{\in}\{r,b\}) & \wedge & (T \vee \mathbf{Y}{\in}\{r,g\}) & \wedge \\ & (F \vee \mathbf{Z}{\in}\{g,b\}) & \wedge & (T \vee \mathbf{Z}{\in}\{r,b\}) & \wedge & (T \vee \mathbf{Z}{\in}\{r,g\}) & \wedge \\ & (\mathbf{Y}{\in}\{g,b\} \vee \mathbf{Z}{\in}\{g,b\}) & \wedge & (\mathbf{Y}{\in}\{r,b\} \vee \mathbf{Z}{\in}\{r,b\}) & \wedge & (\mathbf{Y}{\in}\{r,g\} \vee \mathbf{Z}{\in}\{r,g\}) & \end{array}$$

$$\begin{array}{rccccc} f_1 = & (\mathbf{Y}{\in}\{g,b\}) & \wedge & (\mathbf{Z}{\in}\{g,b\}) & \wedge & \\ & (\mathbf{Y}{\in}\{g,b\} \vee \mathbf{Z}{\in}\{g,b\}) & \wedge & (\mathbf{Y}{\in}\{r,b\} \vee \mathbf{Z}{\in}\{r,b\}) & \wedge & (\mathbf{Y}{\in}\{r,g\} \vee \mathbf{Z}{\in}\{r,g\}) \end{array}$$

$$A_1 = \{\mathbf{X} \mapsto r\}$$

Unit propagation now forces $\mathbf{Y} \mapsto$ {g,b} and $\mathbf{Z} \mapsto$ {g,b}.

$\mathbf{Y} \mapsto$ {g,b} $\quad \mathbf{Z} \mapsto$ {g,b}

$$\begin{array}{rccccc} f_2 = & (\mathbf{Y}{\in}\{g,b\}) & \wedge & (\mathbf{Z}{\in}\{g,b\}) & \wedge & \\ & (\mathbf{Y}{\in}\{g,b\} \vee \mathbf{Z}{\in}\{g,b\}) & \wedge & (\mathbf{Y}{\in}\{b\} \vee \mathbf{Z}{\in}\{b\}) & \wedge & (\mathbf{Y}{\in}\{g\} \vee \mathbf{Z}{\in}\{g\}) \end{array}$$

$$\begin{array}{rccccc} f_2 = & T & \wedge & T & \wedge & \\ & (T \vee T) & \wedge & (\mathbf{Y}{\in}\{b\} \vee \mathbf{Z}{\in}\{b\}) & \wedge & (\mathbf{Y}{\in}\{g\} \vee \mathbf{Z}{\in}\{g\}) \end{array}$$

$$f_2 = (\mathbf{Y}{\in}\{b\} \vee \mathbf{Z}{\in}\{b\}) \quad \wedge \quad (\mathbf{Y}{\in}\{g\} \vee \mathbf{Z}{\in}\{g\})$$

$$A_2 = \{\mathbf{X} \mapsto r,\ \mathbf{Y} \mapsto \{g,b\},\ \mathbf{Z} \mapsto \{g,b\}\}$$

There is now the choice of branching on $\mathbf{Y}$ or $\mathbf{Z}$. Suppose $\mathbf{Y}$ is chosen. Only the branches $\mathbf{Y} \mapsto$ g and $\mathbf{Y} \mapsto$ b need to be explored since $\mathbf{Y} \mapsto$ r was eliminated by the unit propagation step. Suppose $\mathbf{Y} \mapsto$ g was chosen.

$\mathbf{Y} \mapsto$ g

$$f_3 = (F \vee \mathbf{Z}{\in}\{b\}) \quad \wedge \quad (T \vee \mathbf{Z}{\in}\{g\})$$

$$f_3 = \mathbf{Z}{\in}\{b\} \qquad\qquad A_3 = \{\mathbf{X} \mapsto r,\ \mathbf{Y} \mapsto g,\ \mathbf{Z} \mapsto \{g,b\}\}$$

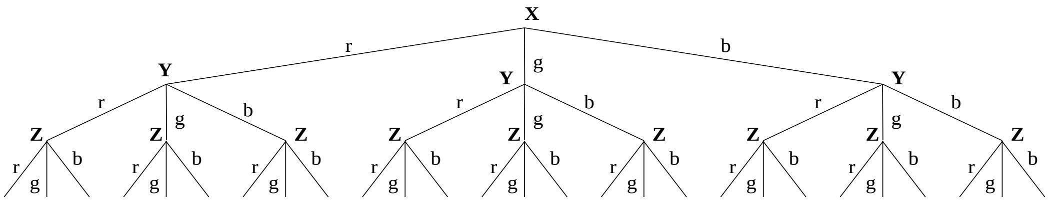
Finally, unit propagation forces $\mathbf{Z} \mapsto \mathrm{b}$.

$\mathbf{Z} \mapsto \mathrm{b}$

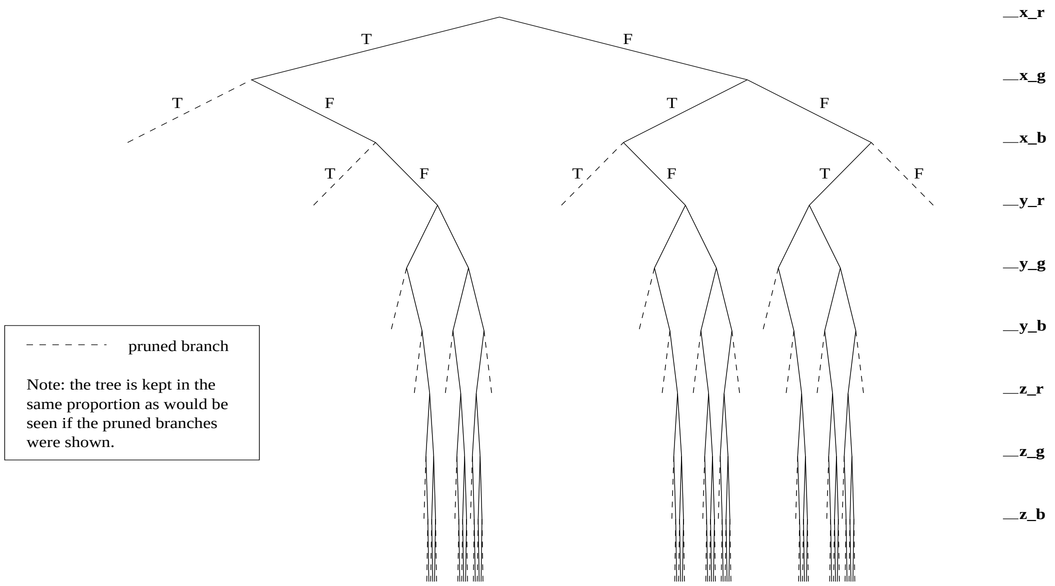
$$f_4 = \quad T$$

$$f_4 = \quad \textit{empty formula} \qquad\qquad\qquad\qquad A_4 = \quad \{\mathbf{X} \mapsto \mathrm{r},\, \mathbf{Y} \mapsto \mathrm{g},\, \mathbf{Z} \mapsto \mathrm{b}\}$$

The empty formula has been found, therefore the formula is satisfied by the assignment A_4.



The non-Boolean search tree, showing the variable and value assignments at each branch.



The search tree of the unary Boolean translation (with assignments omitted from some branches).

The *potential* search space of the translation is much larger (due to non-singular assignments), although not all of it will be *explored*, as described on page 18. The non-Boolean search tree is flatter, because each branch represents a larger commitment, as previously discussed.

3.1.6 Variable and value order

The variable ordering heuristic has the same function as in the Boolean procedure, and I conjecture that it has the same critical effect on it's efficiency. Both the moms and the UP heuristics can be generalised to operate on the non-Boolean search. As in the Boolean procedure, the value order has no effect on the efficiency of the search on unsatisfiable formulas.

3.2 Implementation

The first decision regarding the design was to select a suitable Boolean Davis-Putnam SAT solver on which to base the non-Boolean version. It would, of course, be possible to write the non-Boolean version from scratch but this would require significantly more effort and would be repeating work which has already been done. Many highly-optimised Boolean versions have publicly available source code.

SATLIB[12] provides source code for six Davis-Putnam based solvers: grasp, ntab, POSIT, rel_sat, SATO and satz. Of these, POSIT has the best documentation in Freeman's thesis [9]. In [17], SATO is compared against grasp, POSIT and ntab and is shown to perform well on a variety of problems. Satz (described in [14]) is a relatively new program and was written in the same year as this paper was published (I assume that is why it was not included). When I conducted tests of SATO against satz on the same publicly-available problems as in [17], I found that satz performed well[13].

Problem class	Average run time (seconds)		
	POSIT	SATO	satz
aim-50	0.04	0.01	0.83
aim-100	34.90	0.02	0.21
aim-200	>5000.00	0.06	0.25
bf	>5000.00	0.97	5.20
dubois*	>5000.00	0.03	>5000.00
hanoi	>5000.00	>5000.00	>5000.00
hole	149.17	141.83	87.21
ii8	1.19	0.08	0.56
ii16	** >5000.00	0.64	16.68
ii32	42.75	0.96	** >5000.00
jnh	0.10	0.06	0.33
par8	0.10	0.03	0.11
par16	5.72	77.06	27.61
pret*	>5000.00	0.62	>5000.00
ssa	11.39	0.64	187.02

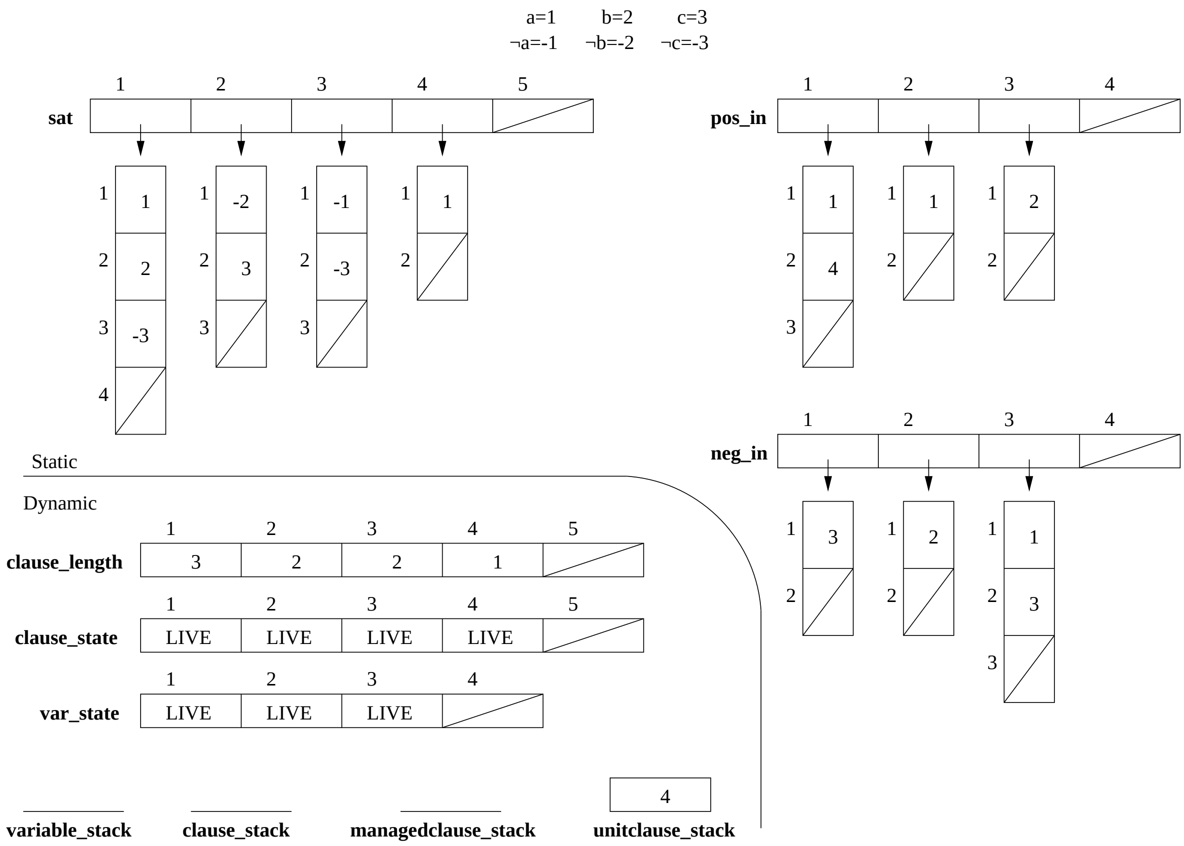
SATO's trie data structures (described in [16]) did not seem to be easily generalisable to handle non-Boolean formulas, and it's dependency-directed backtracking adds a further layer of complexity to the program. On the other hand, the simpler array-based implementations of other solvers are quite easily generalisable. I chose to base my NB solver on satz because I felt that the benefit of it's simplicity outweighed the performance gain from using SATO. Satz is written in 1500 lines of ANSI C code, which is convenient and relatively small compared to some other solvers.

3.2.1 Satz

Satz has a fairly straightforward array- and stack-based implementation. There are two parts of the internal representation, one which remains static throughout the search and one which changes (dynamic). For the formula $(\mathbf{a} \vee \mathbf{b} \vee \neg\mathbf{c}) \;\wedge\; (\neg\mathbf{b} \vee \mathbf{c}) \;\wedge\; (\neg\mathbf{a} \vee \neg\mathbf{c}) \;\wedge\; \mathbf{a}$, the initial state of the data structures is depicted below.

[12] `http://www.informatik.tu-darmstadt.de/AI/SATLIB`

[13] The problems marked * particularly suit dependency-directed backtracking. The performances marked ** are particularly bad on a single instance.



This shows the main data structures, the other parts (such as storing for each variable the current value and unexplored value) are not so important for explaining the operation of the program. The internal representation uses integers to represent variables and clauses; this allows efficient indexing. The static part of the data structure does not change throughout the search, the changes in the state of the formula are flagged in the dynamic part. This reduces the amount of data that needs to be updated during the search and thus reduces the amount necessary to be stored for backtracking.

- The formula is stored in the **sat** array. It is an array of clauses, each clause is an array of literals.
- The **pos_in** and **neg_in** arrays hold, for each variable, the clauses in which it occurs positively (**pos_in**) and negatively (**neg_in**). They allow efficient simplification of the formula after a variable assignment.
- The **clause_length** array holds the current length of each clause.
- The **clause_state** array holds the states of each clause. A clause is DEAD if it has been removed from the formula (i.e. it is True), otherwise it is LIVE.
- The **var_state** array holds the states of each variable. A variable is DEAD if it has been assigned a value, otherwise it is LIVE.

The length of each clause is kept up-to-date during the search, which allows unit clauses to be identified quickly (for efficient implementation of unit propagation). The four stacks hold integer values, corresponding to the variables/clauses that are stored. They are used to store and recall data, to allow efficient backtracking.

- When a variable is assigned a value, the variable is pushed onto the **variable_stack**.
- When a clause has been removed from the formula (i.e. it evaluates to True), it is pushed onto the **clause_stack**.
- When a clause has a literal removed from it (i.e. the literal is False) and the resulting clause is non-unit, it is pushed onto the **managedclause_stack**. If a clause has had two literals removed, it will have two occurrences in the **managedclause_stack**.
- When a clause has a literal removed from it and the resulting clause is unit (i.e. the clause length is 1), it is pushed onto the **unitclause_stack**[14].

The heuristic used by satz is a restricted UP heuristic, as described earlier. This requires additional data structures for the lookahead, but these have been omitted from the explanation for clarity — they only affect the heuristic, and are irrelevant to the rest of the procedure.

Variable assignment and backtracking in satz

When an active variable **x** is assigned True, all active clauses that **x** occurs positively in are removed and all active clauses that **x** occurs negatively in have ¬**x** removed from them. What this amounts to is the following:

1. Set **var_state**[**x**]:=DEAD.
2. Push **x** onto **variable_stack**.
3. For all clauses **c** from **pos_in**[**x**], if **clause_state**[**c**]=LIVE then set **clause_state**[**c**]:=DEAD and push **c** onto **clause_stack**.
4. For all clauses **c** from **neg_in**[**x**], if **clause_state**[**c**]=LIVE then set **clause_length**[**c**]:=**clause_length**[**c**]-1. If **clause_length**[**c**]=0 then backtrack, if **clause_length**[**c**]=1 then push **c** onto **unitclause_stack**, otherwise push **c** onto **managedclause_stack**.

The treatment of assigning a variable False is analogous. Discriminating between unit and non-unit clauses at this stage improves the efficiency of unit propagation. The process backtracks by simply undoing the variable assignments.

1. For all variables **x** on **variable_stack** up to the backtracking point, set **var_state**[**x**]:=LIVE.
2. For all clauses **c** on **clause_stack** up to the backtracking point, set **clause_state**[**c**]:=LIVE.
3. For all clauses **c** on **managedclause_stack** up to the backtracking point, set **clause_length**[**c**]:=**clause_length**[**c**]+1.
4. For all clauses **c** on **unitclause_stack** up to the backtracking point, set **clause_length**[**c**]:=**clause_length**[**c**]+1.

The formula has been satisfied if all clauses have been satisfied (i.e. the **clause_stack** is full), it is unsatisfiable if the search backtracks past the first branch (i.e. the **variable_stack** is empty). Unit propagation is simply a case of finding the only remaining active (LIVE) literal in each clause in the **unitclause_stack** and assigning it's underlying variable True if it is positive and False if it is negative.

[14]Initially, the **unitclause_stack** contains the unit clauses present in the input formula.

The heuristic

The heuristic is implemented with the same operation for assigning variables as detailed above. It effectively explores (to the next level of depth) the subtrees which would result from branching on certain variables. There are separate stacks for variables assigned, shortened non-unit clauses and shortened unit clauses — this is done to maintain a "tentative" state for each variable examined, which can be reset quickly.

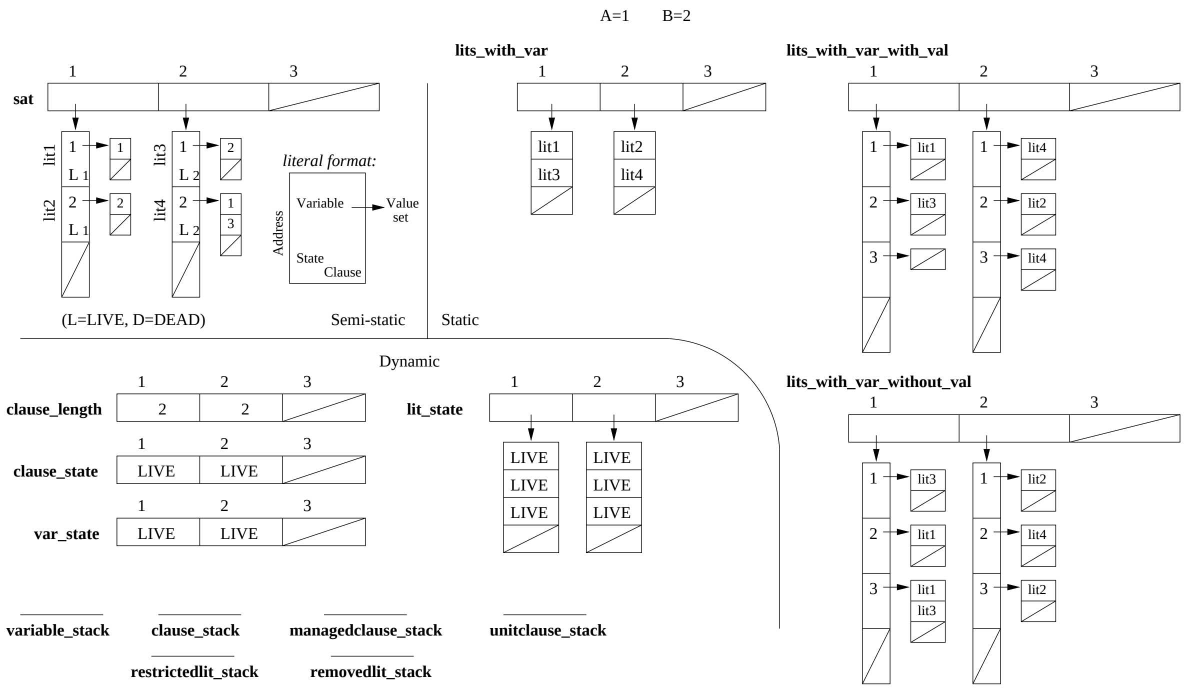
3.2.2 NB-satz

The non-Boolean procedure described on page 24 differs from the Boolean one in a number of features.

1. The domain size is not known at compile-time, so the data structures must not be domain size specific (e.g. **pos_in** and **neg_in** from the Boolean implementation).
2. Each literal may have an arbitrary value set (limited in size by the domain).
3. Individual literals must be able to be eliminated from the formula. This contrasts to removing *all* literals l and *all* clauses $\bar{l} \vee \alpha$, as in the Boolean procedure.

The first two points are relatively trivial to implement and do not pose a problem, although the third is not so easy to implement efficiently. This contrasts heavily to the Boolean implementation where a single flag for each variable was sufficient (**var_state**). This point is discussed later.

The same basic data structures as satz are used, but extra ones are introduced to follow the procedure on page 24. The static representation is altered to allow efficient indexing, with pointers to literals being used instead of array indices. Two new stacks are introduced to hold restricted value sets (**restrictedlit_stack**) and individual literals eliminated from the formula (**removedlit_stack**). For the formula $(\mathbf{A}\in\{1\} \vee \mathbf{B}\in\{2\}) \wedge (\mathbf{A}\in\{2\} \vee \mathbf{B}\in\{1,3\})$, the initial state of the data structures is shown below.



Each entry in the **sat** array is now a 4-tuple, holding the variable, it's value set, the clause it occurs in and the individual literal state — the only part of the **sat** data structure that is dynamic.

The operation of the dynamic part of NB-satz is identical in operation to that in satz, except for the addition of **lit_state**, **restrictedlit_stack**, **removedlit_stack** and the addition of individual literal states. The value set for a particular variable is restricted by flagging it's state in the **lit_state** array, similarly to how a clause is removed. Each of these restrictions is pushed on the **restrictedlit_stack**, so that they can be restored when backtracking. Individual literals are removed by setting their state and pushing the literal pointer (address) onto the **removedlit_stack** (for backtracking). The individual literal state is necessary from requirement 3 above. Although these three data structures for the states of literals create redundancy (**var_state**, **lit_state** and the individual literal state), having all three improves time efficiency.

Since the procedure is operating on individual literals, it is necessary to have the **pos_in/neg_in** arrays replaced by ones that index individual literals rather than clauses (as in the Boolean implementation). Some operations still operate on clauses, so each literal also stores it's clause. The extra array "**lits_with_var**" is used for unit propagation.

Variable assignment and backtracking in NB-satz

When an active variable $\mathbf{X}$ is assigned the value $\mathbf{a}$, all active clauses that contain $\mathbf{X}\in\{\mathbf{a},\ldots\}$ are removed. All other active clauses that contain the literal $\mathbf{X}\in\{\ldots\}$ have $\mathbf{X}\in\{\ldots\}$ removed from them.

1. Set **var_state**[**X**]:=DEAD.
2. Push **X** onto **variable_stack**.
3. For all clauses **c** from **lits_with_var_with_val**[**X**][**a**]->clause, if **clause_state**[**c**]=LIVE then set **clause_state**[**c**]:=DEAD and push **c** onto **clause_stack**.
4. For all clauses **c** from **lits_with_var_without_val**[**X**][**a**]->clause, if **clause_state**[**c**]=LIVE then set **clause_length**[**c**]:=**clause_length**[**c**]-1. If **clause_length**[**c**]=0 then backtrack, if **clause_length**[**c**]=1 then push **c** onto **unitclause_stack**, otherwise push **c** onto **managedclause_stack**.

As before, the process backtracks by simply undoing the variable assignments.

1. For all variables **X** on **variable_stack** up to the backtracking point, set **var_state**[**X**]:=LIVE.
2. For all clauses **c** on **clause_stack** up to the backtracking point, set **clause_state**[**c**]:=LIVE.
3. For all clauses **c** on **managedclause_stack** up to the backtracking point, set **clause_length**[**c**]:=**clause_length**[**c**]+1.
4. For all clauses **c** on **unitclause_stack** up to the backtracking point, set **clause_length**[**c**]:=**clause_length**[**c**]+1.

At each branch in the search, only the subtrees corresponding to currently active values are explored (i.e. for the variable $\mathbf{X}$ with active values $\{\mathbf{a}_1,\mathbf{a}_2,\ldots\mathbf{a}_n\}$ and inactive values $\{\mathbf{b}_1,\mathbf{b}_2,\ldots\mathbf{b}_n\}$, only $\mathbf{X}\mapsto\{\mathbf{a}_1\}$, $\mathbf{X}\mapsto\{\mathbf{a}_2\}$, $\ldots$, $\mathbf{X}\mapsto\{\mathbf{a}_n\}$ are explored).

Unit propagation

Unit propagation is not as simple as in the Boolean procedure, since it may entail a partial assignment. Each active unit clause $\mathbf{X}\in\{\mathbf{a}_1,\mathbf{a}_2,\mathbf{a}_3,\ldots\}$ in the **unitclause_stack** is propagated by the following method.

1. For all values $\mathbf{a} \in (D\backslash\{\mathbf{a}_1,\mathbf{a}_2,\mathbf{a}_3,\ldots,\mathbf{a}_n\})$, if **lit_state[X][a]**=LIVE then set **lit_state[X][a]**:=DEAD and push (**X**,**a**) onto **restrictedlit_stack**.
2. For all literals **l** in **lits_with_var[X]**, if **l**->state=LIVE and **clause_state[l**->clause]=LIVE then:
    (a) Let **c**=**l**->clause.
    (b) If $\mathbf{l}=\{\mathbf{a}_1,\mathbf{a}_2,\mathbf{a}_3,\ldots,\mathbf{a}_n\}$ then set **clause_state[c]**:=DEAD, push **c** onto **clause_stack**.
    (c) If **l**={} then set **l**->state:=DEAD and push **l** onto **removedlit_stack**. Set **clause_length[c]**:=**clause_length[c]**-1. If **clause_length[c]**=0 then backtrack, if **clause_length[c]**=1 then push **c** onto **unitclause_stack**, otherwise push **c** onto **managedclause_stack**.

The internal representation of literals

All literals are internally represented positively. This can be inefficient if the formula is most concisely expressed using negative literals (e.g. graph colouring problems), because the internal data structures are larger and more state needs to be maintained during the search. The Boolean translation can have a more compact representation and avoid this problem to an extent, because the more concise translation can be used (see page 20).

Non-Boolean heuristics

In the chosen branching method, the number of branches at each node increases with domain size (each node has more subtrees). Because of this, if the "goodness" of a branch is determined by it's worst subtree, this has a negative effect on the NB procedure. By grouping more sub-searches into one branching point, the best branch is less likely to be very much better than an average branch.

This affects the choice of variable ordering heuristic — if the best branch is not much better than a good one, a heuristic that is rough but quick may outperform a more accurate but slower one. For this reason, I implemented a version of NB-satz with the moms heuristic as well as one with the UP heuristic.

Let $n_2(l)$ be the number of times literal l occurs in binary clauses
$n_3(l)$ be the number of times literal l occurs in ternary clauses

$$w(l) = (5 \times n_2(l)) + n_3(l)$$

The basic non-Boolean moms heuristic is a straightforward generalisation of the Boolean balanced and weighted version. Over all active variables X, with active values $a_1, a_2, \ldots a_n$, the basic moms heuristic selects the variable X which maximises $min\,(w(X \in \{a_1, \ldots\}), w(X \in \{a_2, \ldots\}), \ldots, w(X \in \{a_n, \ldots\}))$.

Recall that for each variable examined, the UP lookahead step explores the effects of assigning all active values to that variable. The chosen NB branching method therefore forces the cost of examining a single variable to be greater as domain size increases. Combined with the previous point (the *best* branch is not much better than a *good* one), this reduces the efficiency of the UP heuristic.

The lookahead step is easily generalised; a variable X with active values $\{a_1, a_2, \ldots, a_n\}$ in a formula f is examined by the following procedure:

- The formula p_k is obtained by performing unit propagation on $f \wedge X \in \{a_k\}$.
- The individual weights of each $X \mapsto k$ are given by $w(X \mapsto k) = diff(p_k, f)$, where $diff(a, b)$ is the number of clauses that are shorter in a than in b.
- The overall weight given to the variable is $H(X) = \frac{1}{k}\Sigma_{i=1}^{k} w(X \mapsto k)$.

I found that this weight function performed better than the straight generalisation of Freeman's weight function, which is $H(X) = 1024 \times \Pi_{i=1}^{k} w(X \mapsto k) + \Sigma_{i=1}^{k} w(X \mapsto k)$, which ignores the number of active values. I dropped the Π term because it didn't improve the heuristic (it is unlikely to be significant for large domains, since at least one branch is likely to be poor). I added the $\frac{1}{k}$ factor because it alters $H(X)$ to favour smaller sets of active values — this produced the largest performance gain I could obtain from tweaking the UP heuristic.

This non-Boolean generalisation of the UP heuristic was less effective on increasing domain sizes. To try to decrease the search time, I tried to restrict it in the same way as is done with the Boolean version. I kept the same concept, but generalised it to be "at least i occurrences, at least j with each active value". For various domain sizes, I tried various values for i and j, but for larger domains, I did not observe performance gains such as those seen in the Boolean procedure [14]. After looking at the search trees and the number of variables examined at each node, I thought of a more relaxed restriction method: "at least i occurrences, at least j with each value from a subset of k active values". I tried many values for i, j and k for various domain sizes but could not find any restrictions that performed better than the pure UP heuristic (unrestricted) on large domains (size > 4). The implementation used for testing uses the pure UP heuristic. I conducted these tests on random formulas, as in [14].

Refining the moms heuristic

The moms heuristic attempts to select the most constrained variable based on syntactic analysis of the formula. The Boolean procedure considers the number of occurrences of each variable and the size of the clauses they occur in (and attempts to maximise the number of occurrences and minimise clause size). The non-Boolean procedure introduces another varying factor — the size of the value set.

The unit propagation step can be applied to any literal, regardless of the size of it's value set. Although a unit clause containing a literal with a large value set may actually produce more unit propagation than one with a smaller value set, it is less likely to do so (variables with smaller value sets are intuitively more tightly constrained). It is the task of the heuristic to *estimate* the effect of branching on a particular variable, so this estimation should take into account the value set size of literals.

Implementing this idea is non-trivial because the value set size of each literal in the formula must be known. Whether this is maintained during the search or determined each time the heuristic is called, the time overhead is quite significant. A more feasible method is to consider the number of *active* values associated with each variable (i.e. the **lit_state** is LIVE), which requires time dependent on the number of variables and values, not the formula size. This is a rougher estimate, but can be computed quickly.

Let $n_2(l)$ be the number of times literal l occurs in binary clauses
$n_3(l)$ be the number of times literal l occurs in ternary clauses

$$w(l) = (5 \times n_2(l)) + n_3(l)$$

Over all active variables X, with active values $a_1, a_2, \ldots a_n$, the refined moms heuristic selects the variable X which maximises $\frac{1}{n}\left[min\left(w(X \in \{a_1, \ldots\}), w(X \in \{a_2, \ldots\}), \ldots, w(X \in \{a_n, \ldots\})\right)\right]$. This favours variables with smaller sets of active values. I found that this performed better than the basic moms heuristic, and reduced the average search time by a factor of about 20% on hard random 3-SAT formulas.

Other points

A literal with n values is internally represented as a variable and a value set comprising n values; an alternative to this is to represent it as n variable-value pairs. This implementation makes the unit propagation step very difficult, because the clause length can no longer be used to indicate a unit clause. Eliminating a clause $\mathbf{X}\in\{a_1,a_2,\ldots,a_n\} \vee \ldots$ because $\{a_1,a_2,\ldots,a_n\}$ is equal to the current partial assignment of $\mathbf{X}$ (part of the unit propagation step) also becomes difficult.

The non-Boolean unit propagation step is fundamentally different from the Boolean unit propagation step, with respect to the implementation. If the clause $\mathbf{X}\in\{a,b,c\}$ is propagated, the literal $\mathbf{X}\in\{c,d\}$ will be removed, but $\mathbf{X}\in\{a,b\}$ will not. Therefore each *individual* literal must be able to be removed from the formula, independent of all other literals.

This can be achieved two ways: either keep a state for each variable-value pair, or keep a state for every literal in the formula — NB-satz does both of these, for time efficiency. The state of each variable-value pair (**lit_state**) is necessary so that the active values (those which need to be explored at each branch) can be determined quickly. The non-Boolean procedure could have been implemented without the individual literal state, but then the state of each literal (often required to be known in the procedure) would only be able to be determined performing many comparisons between the literal's value set and the states of each variable-value pair. My chosen implementation method sacrifices increased memory (and state) for a faster search.

The non-Boolean UP heuristic does not discriminate between a unit propagation that assigns $\mathbf{X} \mapsto \{a\}$ and one that assigns $\mathbf{X} \mapsto \{a,b\}$. This is a potential weakness — the former is intuitively more preferable since it eliminates a variable from the formula.

3.3 Program verification

For the verification, I assumed that satz was correctly implemented and always gives the correct solution (satisfiable or unsatisfiable). The first task was to show that NB-satz agreed with satz as to whether a formula is satisfiable or unsatisfiable. To do this I repeatedly generated a random non-Boolean formula, solved it with NB-satz and solved it's Boolean translation with satz. A simple batch file did this and halted if they gave a different result. I chose to repeat this procedure using hard[15] random formulas over a number of domain sizes and formula sizes, to maximise the chance of finding a discrepancy. After a few thousand iterations no discrepancy was found. Both the moms implementation and the UP implementation were tested in this manner, which further increased my confidence in a correct

[15] Generated at the phase transition, explained later.

implementation.

If NB-satz and satz had the same behaviour on formulas with domain size 2, this would aid verification immensely since both searches could be traced and compared — it would be extremely unlikely that there there was an error if both searches were the same. Since it was not possible for me to find an efficient restriction on the unit propagation heuristic, the original implementation of satz could not be directly compared to NB-satz in this way (either the moms or the UP implementation) because the different heuristic would produce a different variable ordering.

I decided that this was a valuable test so I adapted satz to use the same moms heuristic as in NB-satz and found that both produced the same search on all formulas I tested them on. A small thing that confused me was that initially it gave the same results (i.e. number of branches, backtracks etc.) for unsatisfiable formulas but not the same search, while for satisfiable formulas neither the results or the search was always the same. I then realised that the converter used the binary encoding $\mathbf{X}\in\{0\} = \neg\mathbf{x}$ and $\mathbf{X}\in\{1\} = \mathbf{x}$, and that the Boolean value order (T first, then F) was opposite to how I implemented it in NB-satz (0, 1, 2, ...). When I reversed the value order in satz, all the searches and solutions were identical. The UP implementation of NB-satz differs from the moms version only in the heuristic (the variable order), so these tests gave me confidence that both were correct.

When a formula is determined to be satisfiable, the solution is checked by a simple procedure that puts the values found back into the formula and tests if all clauses are satisfied, an error message appears if this is not so. This simple step ensures that all satisfying assignments output by the program actually satisfy the formula (i.e. it confirms the validity of each solution produced). The program never produced a "duff" assignment on any of it's runs. Of course all these tests are necessary but not sufficient to show a valid implementation, i.e. they do not *prove* that NB-satz is correct.

3.4 Program evaluation

To evaluate the performance of NB-satz, it has to be tested on a number of formulas. Obviously a wide range of test formulas would be good, to give as much information as possible, but the formulas should be related (controlled) in some way, so as to enable conclusions to be drawn from the data. I based the decision of which problems to use as a test on these requirements.

3.4.1 Randomly generated formulas

Many SAT search procedures are tested on randomly generated formulas because they can be easily generated and with a high degree of control. The number of variables, number of clauses and the domain size can be varied, and the formulas can be generated in such a way so that they are hard to solve (few solutions). There are disadvantages though — the formulas do not relate to the encoding of any real-world problems (whose formulas may be very different) and they are all similar in structure. This controllable similarity can be a good thing for benchmarking, but a SAT solver is not of practical use if it only performs well on random formulas.

Boolean random SAT formulas are commonly generated using the constant clause length model (k literals per clause — called k-SAT). Each clause in a k-SAT formula is generated by choosing k distinct variables at random from the set of variables, and negating them with probability $\frac{1}{2}$. Random 3-SAT is usually used to test solvers because it is NP-complete. Non-Boolean random formulas are generated in a similar way, but a constant literal set size is imposed. A NB (k,l)-SAT clause is generated by picking k distinct variables at random, and l distinct values at random for each of the variables. Frisch and Peugniez [10] use the model where l is equal to half the domain size. This stays true to the Boolean random formula model when the domain size is two. I chose this method (as opposed to $l = 1$ for instance) because it puts a relatively equal constraint on each variable, independent of domain size. Examples of literals generated in this way are given below.

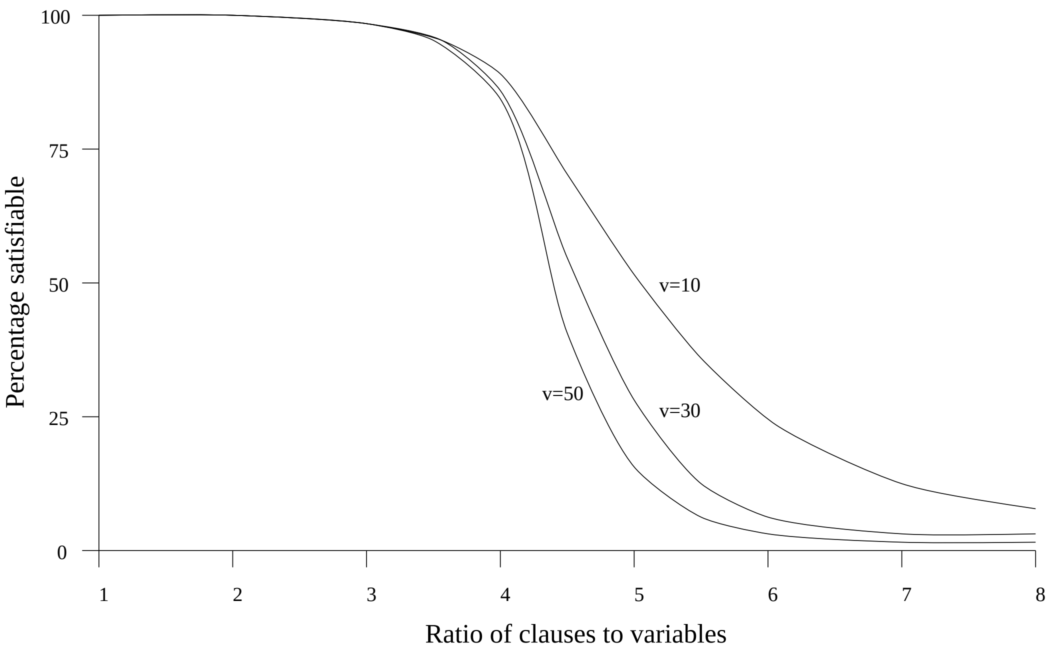
Domain = {1,2}	Domain = {1,2,3,4}	Domain = {1,2,3,4,5,6,7,8}
A∈{1}	**A**∈{1,2}	**A**∈{3,4,6,8}
B∈{2}	**B**∈{2,3}	**B**∈{1,4,5,7}
C∈{1}	**C**∈{2,4}	**C**∈{2,3,5,7}
D∈{1}	**D**∈{1,4}	**D**∈{1,2,5,6}

The phase transition in random formulas

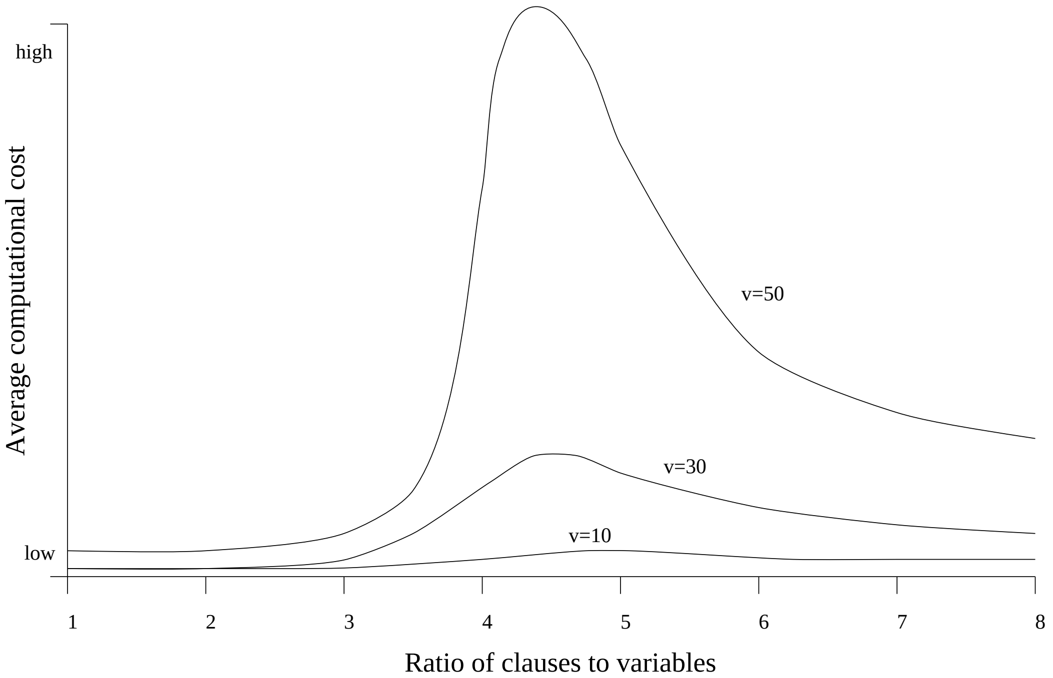
In 1979, Goldberg [11] showed that the Davis-Putnam procedure ran in polynomial time in the average case. He was using randomly generated formulas, but these were easy to solve, as shown by Franco and Paull in 1983 in [8]. Around this time, there was an intellectual awakening regarding the phase transition in SAT. Taking the ratio of clauses to variables (c/v) as a measurement, the satisfiable and unsatisfiable problems can be grouped. The satisfiable formulas tend to occur with low values of c/v, the unsatisfiable ones tend to occur with a higher c/v ratio.

If we keep v constant and generate a number of random formulas with varying c, this behaviour can be explained. The formulas with few clauses (low c/v) are under-constrained — many assignments are likely to satisfy the formula because there are few clauses to be violated. Likewise, formulas with many clauses (high c/v) are over-constrained and most assignments violate at least one clause, since there are more to violate.

The graph below shows the trend of satisfiability against c/v. The phase transition is the region of the curve where the percentage of formulas satisfiable goes from very high to very low. It is described as a phase transition because it mimics the knife-edge characteristics of many natural phenomenon such as water freezing. 3-SAT is the smallest k-SAT problem that is NP-complete [2], so is commonly used to test SAT solvers. The phase transition in 3-SAT has been studied much and the point where 50% of the formulas generated are satisfiable occurs at around $c/v = 4.3$.



As the ratio of clauses to variables changes, the amount of computational effort required to solve the formulas changes. For under-constrained formulas, the tree holds many solutions and one is quickly found, so not much of the tree actually needs to be explored. For over-constrained formulas, many contradictions are found early in the search, so much of the tree can be pruned away. The most difficult formulas for the Davis-Putnam method occur at the phase transition, where the formulas are only just satisfiable or very nearly satisfiable. For these formulas there are not many solutions, if any, so the search cannot stop early. Nor can it prune away much of the tree because there are no contradictions high in the tree. It is widely accepted that random formulas used for testing SAT solvers should be generated at the phase transition, as they are the more difficult problems.



The same easy-hard-easy characteristic is shown in randomly generated non-Boolean formulas. The random formulas used to test NB-satz should be generated at the phase transition — it was the task of early experiments to locate this point for various formula parameters.

Generating formulas on the phase transition

I wrote a small program which repeatedly generates random formulas (by calling another program) and solves them. Throughout this process, one of the generation parameters is systematically varied, to home in on the point where 50% are satisfiable. The number of variables is kept constant and the number of clauses is varied, as in [10]. The motivation for this is to keep the search space size constant for each domain size by choosing the number of variables in each formula; the number of clauses is then dictated by this.

Search space size

The size of the search space of each non-Boolean formula was kept constant for all domain sizes (the Boolean translation may, of course, increase the search space). For domain size d and v variables, the search space size is d^v (each of the v variables may have d values). I chose to keep $d^v = 2^{60}$ as these formulas are large enough to be difficult, without being too large to hamper testing. The final parameters were determined as:

Domain size	Number of variables	Number of clauses
2	60	261
4	30	280
8	20	294
16	15	302
32	12	307

3.4.2 Graph colouring

Graph colouring problems were described earlier, their non-Boolean encoding is of interest because it is intrinsically non-Boolean. A number of benchmarks are available from various sources, including the DIMACS[16] web site. These are not entirely ideal for testing NB-satz because they are mainly too large, and the problems which differ in domain size (number of colours) also differ in other aspects. Control over the steps in domain size would be an advantage, to allow an even spread of problems, therefore I considered generating my own graphs.

Randomly generated graph colouring problems

When a graph colouring problem is translated into a satisfiability problem (Boolean or non-Boolean), the number of colours to be used must be specified. The problem is then not to colour the graph in the smallest number of colours, but to determine whether it can be coloured with k colours. To ensure a non-trivial problem, k must be set to the minimal number of colours for each graph. Therefore the minimal colouring number needs to be known before the translation to SAT.

An iterative approach using the Davis-Putnam procedure was used to find the minimal number of colours, but this was a little awkward, because the Davis-Putnam procedure takes much longer to

[16] `http://dimacs.rutgers.edu/Challenges`

terminate on unsatisfiable graph colouring formulas than satisfiable ones. For any graph, the search only *needs* to explore one subtree of the root node, since the first commitment (choice of node and colour) can never commit to a tree devoid of a solution. For unsatisfiable graph colouring problems, the Davis-Putnam search does much unnecessary work and actually explores all colours for the first choice of node. I attempted to use a special-purpose graph-colouring algorithm to determine the minimal colouring, but the only ones I could find were incomplete and could not guarantee that the colouring they produced was minimal. I was therefore confined to generating smaller graphs than I would have liked, since determining the minimal number of colours was time-consuming using Davis-Putnam.

I decided to generate graphs with randomly assigned edges, other methods (e.g. the k-colourable method) can tend to generate "easy" graphs. I used Joe Culberson's quasi-random graph generator[17] because it is simple to use and outputs graphs in the DIMACS format — readable by the program which translates them into SAT problems. I kept the number of nodes constant at 30 and varied the number of edges (the sparsity/density of the graph), therefore altering the minimal colouring. For the tests, I selected a number of graphs with a range of domain sizes (minimal number of colours). I chose not to use any unsatisfiable graph colouring problems because the DP search is very inefficient for these, particularly on larger domain sizes (the reason for this is detailed above).

Benchmark problems

Most of the available graph colouring benchmarks are too large to solve feasibly (either with NB-satz or satz). The smaller graphs consist of randomly generated graphs, n-queens problems represented as graphs, geographical graphs and book graphs. The geographical graphs represent American cities — if two cities (nodes in the graph) are within k miles of each other then there is an edge between them. The collection is made up from the same set of cities for different values of k. The nodes in the book graphs are the characters in the book, edges represent two characters coming into contact with each other.

The advantage of using benchmark problems is that the minimal colouring is usually known. Non-randomly generated graphs may have different structures to randomly generated graphs, so they give a wider range of test formulas. It would have been nice to have some real-world problems, such as register allocation or timetabling, but I could not find any that were suitable.

I decided to test on the following benchmarks:

- miles250 and miles500
- n-queens (5x5, 6x6, 7x7, 8x8)

3.4.3 Planning

Planning is an important problem in computer science, and is concerned with finding a way to achieve a certain goal. The plan is decided in the context of a world modelled by it's state (e.g. the position of objects in a room) and a number of possible actions (e.g. moving a box, pushing a switch). Actions have preconditions and effects — certain actions can only take place at certain times and all actions may affect the state of the world. The task is to find a sequence of actions that achieves the goal, or more specifically the optimal sequence of actions (the shortest plan). Planning has important practical applications.

[17] `http://web.cs.ualberta.ca/~joe/Coloring/index.html`

Translating a planning problem into Boolean SAT and solving with a general-purpose SAT solver has been shown to outperform specialised planning algorithms. Matthew Slack, a student at the university, is currently doing a project on translating planning problems into non-Boolean SAT (effectively a stepping-stone between the original encoding and the Boolean SAT encoding). I gave him NB-satz and he gave me a number of planning problems in the non-Boolean encoding. This is a good source of non-random formulas with which to test NB-satz.

3.4.4 Testing procedures

The test problems were solved once each — because the procedures are deterministic, they perform exactly the same search each time they are run on a problem. I collected all the results from each class of problem in a file which was then processed to generate means etc. This was done to keep all the results from each individual formula, so that they could be examined if necessary. I ran all the tests on a Silicon Graphics workstation with a 133MHz MIPS processor and 64Mb RAM.

I decided to test against the Boolean solver satz, which uses an efficiently restricted UP heuristic. This is not entirely "fair" because both the non-Boolean procedures that I developed use inferior heuristics. I decided against testing on a dumbed-down version of satz (a version with a moms heuristic and one with a pure UP heuristic) because I felt that although this may give more comparable results, it missed the full picture — the Boolean solver has an efficient restriction on the UP heuristic, but I failed to discover an efficient non-Boolean generalisation. I felt that this was an important disadvantage of the non-Boolean procedure and it should be reflected in the results.

Results

The results are presented in the following tables and graphs. I use graphs where appropriate because they show at a glance the various trends in the data and ease visualisation.

The "formulas size" measurement follows the same convention as in [10]; the size of a non-Boolean formula is the number of values it contains (i.e. the sum of the sizes of each literal's value set), the size of a Boolean formula is the number of literals it contains. This measure is used to compare the sizes of the original non-Boolean formulas with their Boolean translation, to give an indication of the enlargement introduced by having to enforce singular assignments in the input formula. The non-Boolean graph colouring and planning formulas are measured by their most concise representation, i.e. negative literals are used where it makes the formula smaller. It should be remembered that this measure may not reflect the size of the internal data structures.

The number of times the search branches and backtracks are included to indicate the size of the search tree. For domain size d, A non-Boolean branch produces $\leq d$ subtrees, so the proportion of branches to backtracks is typically lower for larger domains.

The binary encoding is only included for the graph colouring experiments; it is omitted for the others because it was unfeasible — the size of the translated formula was extremely large.

The methods tested are: non-Boolean with moms heuristic (NB *moms*), non-Boolean with UP heuristic (NB *UP*), unary translation (U), unary-binary translation (UB) and binary translation (B).

4.1 Randomly generated formulas

The results from the randomly generated formulas are split up into satisfiable and unsatisfiable formulas, to highlight the differences between them. Formulas were generated at the phase transition and solved by each of the methods. Only the results for the first fifty satisfiable formulas and the first fifty unsatisfiable formulas were kept (to keep the sample size constant).

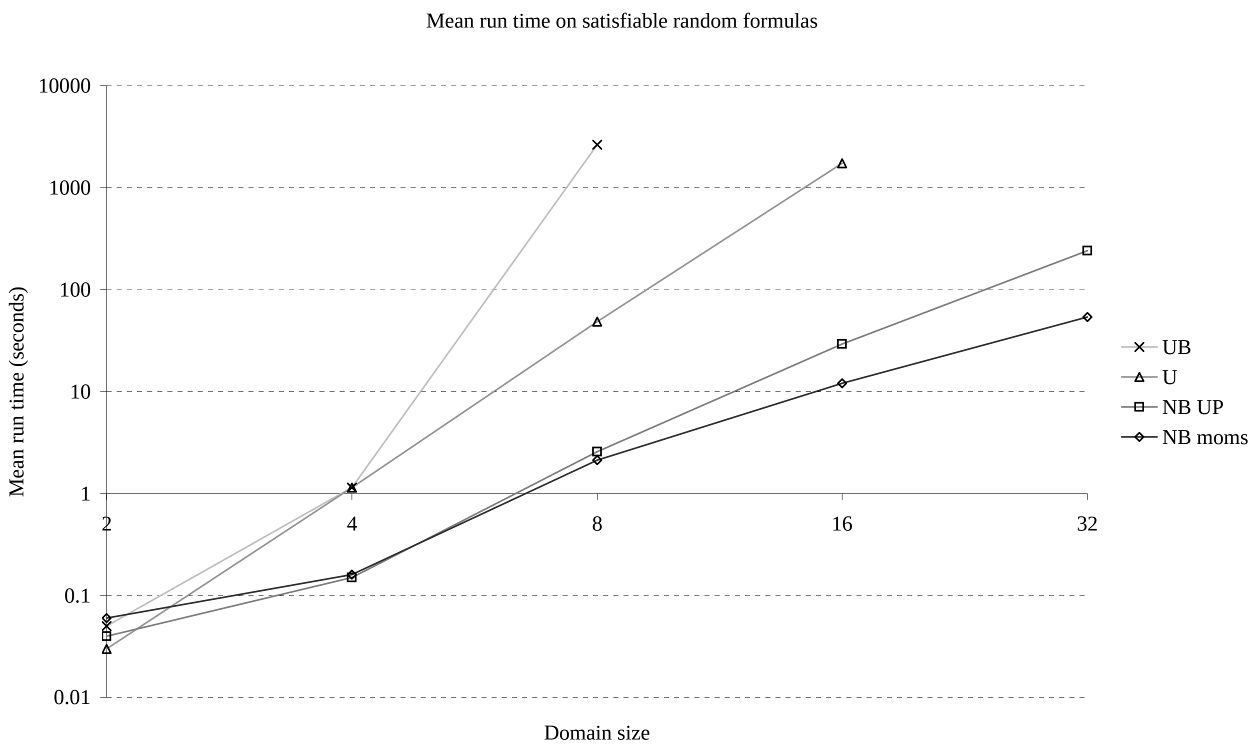
These formulas were generated using the previously described method. They are all positive formulas, so the ALO clauses are not included in the U and UB translations.

Both the x and y scales on the graphs are logarithmic, to increase readability.

Some methods were particularly slow, so only ten formulas were tested. These are marked with *.

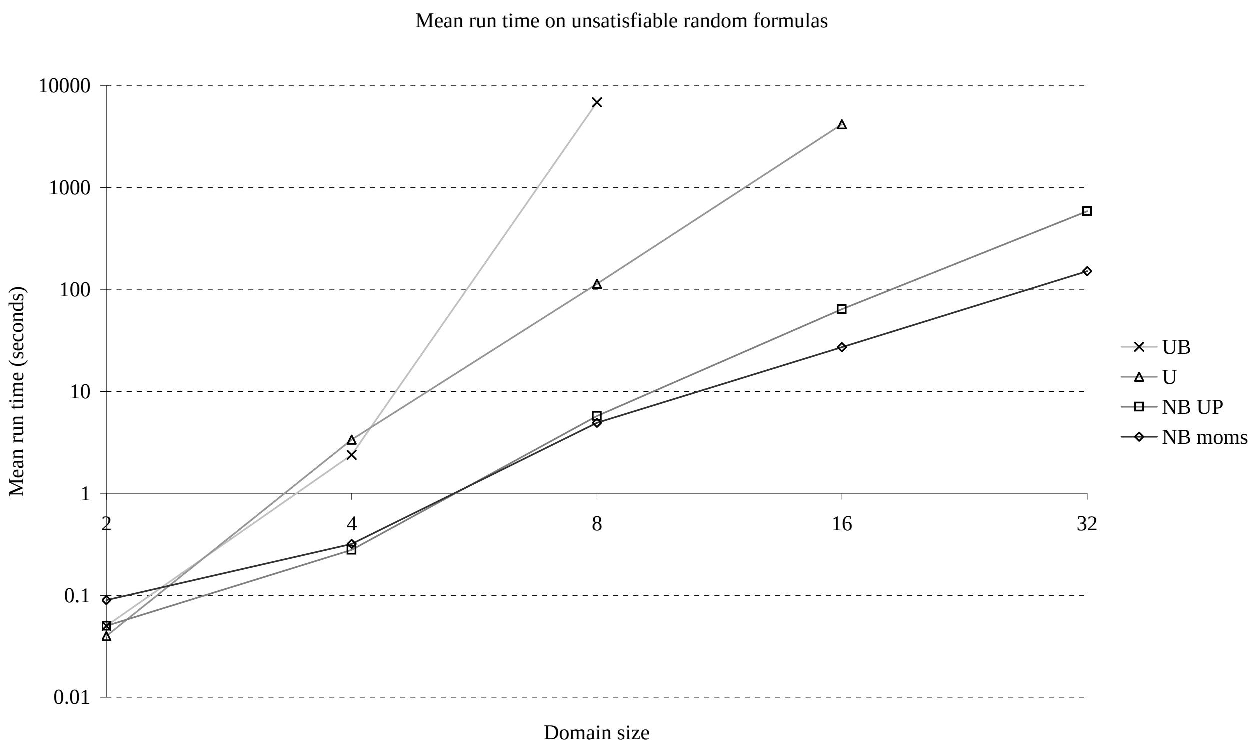
4.1.1 Satisfiable formulas

Problem	Method	Formula size	Mean branches	Mean backtracks	Mean unit	Mean pure	Mean run time (sec)
Domain size 2 60 variables 261 clauses	NB *moms*	783	27.36	21.82	422.38	16.32	0.06
	NB *UP*		8.08	3.44	74.86	8.26	0.04
	U	903	10.32	2.50	164.44	16.48	0.03
	UB	1023	10.40	2.34	248.48	22.98	0.05
Domain size 4 30 variables 280 clauses	NB *moms*	1680	93.12	133.72	3850.68	5.92	0.16
	NB *UP*		9.38	16.92	391.16	1.52	0.15
	U	2040	1058.94	557.18	9024.80	20.60	1.14
	UB	2160	1228.60	652.96	18626.20	17.68	1.14
Domain size 8 20 variables 294 clauses	NB *moms*	3528	1102.10	2816.58	86829.24	2.38	2.13
	NB *UP*		43.86	221.52	7259.96	0.52	2.56
	U	4648	10929.06	5642.52	101305.50	7.24	48.31
	UB*	4488	1940551.20	1035063.10	36138227.20	7010.20	2624.50
Domain size 16 15 variables 302 clauses	NB *moms*	7248	3940.38	12754.08	630039.62	0.40	12.00
	NB *UP*		123.56	1200.50	69570.32	0.04	29.18
	U*	10848	194440.00	99145.10	1911379.60	8.90	1736.04
	UB*	8448	—	—	—	—	>10000.00
Domain size 32 12 variables 307 clauses	NB *moms*	14736	9659.02	41231.14	2917633.60	0.28	53.77
	NB *UP*		312.86	5329.20	514793.98	0.12	240.92
	U*	26640	—	—	—	—	>10000.00
	UB*	18576	—	—	—	—	>10000.00



4.1.2 Unsatisfiable formulas

Problem	Method	Formula size	Mean branches	Mean backtracks	Mean unit	Mean pure	Mean run time (sec)
Domain size 2 60 variables 261 clauses	NB *moms*	783	44.94	45.94	817.26	10.90	0.09
	NB *UP*		7.66	8.66	89.92	3.16	0.05
	U	903	12.90	7.22	310.26	6.86	0.04
	UB	1023	12.22	6.78	460.32	8.70	0.05
Domain size 4 30 variables 280 clauses	NB *moms*	1680	190.48	288.56	8305.40	3.94	0.32
	NB *UP*		14.16	39.40	796.82	0.48	0.28
	U	2040	3094.56	1637.52	26240.84	39.84	3.35
	UB	2160	2609.58	1393.32	39062.70	20.86	2.38
Domain size 8 20 variables 294 clauses	NB *moms*	3528	2550.72	6592.32	205104.10	2.66	4.93
	NB *UP*		92.32	499.80	16240.52	0.16	5.72
	U	4648	25542.32	13204.60	238834.94	20.06	113.50
	UB*	4488	5098866.40	2724622.80	94608883.20	18394.30	6848.84
Domain size 16 15 variables 302 clauses	NB *moms*	7248	8866.50	28787.58	1429531.70	0.66	27.06
	NB *UP*		274.42	2710.36	157733.88	0.06	63.77
	U*	10848	452363.60	231142.30	4451014.00	28.60	4151.00
	UB*	8448	—	—	—	—	>10000.00
Domain size 32 12 variables 307 clauses	NB *moms*	14736	26550.30	114463.22	8196184.10	0.14	150.08
	NB *UP*		756.08	13119.96	1293095.60	0.00	583.72
	U*	26640	—	—	—	—	>10000.00
	UB*	18576	—	—	—	—	>10000.00



Both the solution times and the search tree size increase for all the methods as domain size increases. This trend is also shown to occur with NB-walksat in [10].

The results for the satisfiable formulas are very close to those for the unsatisfiable formulas, except the mean run time is roughly half, corresponding to solutions lying roughly half way through the search space on average. NB *moms* outperforms NB *UP* on domain size 8 and above, although this is a "brute-force" win, since the search tree size (indicated by the number of branches) is clearly worse — the heuristic is less accurate. It seems that it is not worth the time needed to work out the best[18] branch, but a good one will do if it is found quickly.

The UB translation performs particularly poorly, the growth of which is the worst of all the methods. I would suggest that this is because it introduces more variables into the translated formula. The U translation performs better than UB, but much worse than both NB *moms* and NB *UP* as domain size increases. Looking at the number of branches and backtracks for the Boolean encodings, it can be seen that they explore more of the search space. I infer from this that using the Boolean heuristic on the translation is less accurate than using the non-Boolean heuristic on the non-Boolean formula.

Both NB *moms* and NB *UP* show a bump in run time in going from domain size 2 to domain size 16. It seems that these formulas increase rapidly in difficulty between around domain size 4 and 8, after which the increase is not so large. I do not have an explanation for this phenomenon but it is clearly not a coincidence because it happens with both heuristics and both satisfiable and unsatisfiable formulas. More experiments on domain sizes between 4 and 8 would be interesting and may shed light on the underlying cause.

Within the program, I recorded the unit propagation with two separate counts. One holds the number of unit propagations $\mathbf{X} \mapsto \{a_1\}$, one holds the unit propagations $\mathbf{X} \mapsto \{a_1, a_2, \ldots, a_n\}$. These are referred to as full- and semi-unit propagation, respectively. The total amount of all unit propagation is the sum of these values, the mean of which is displayed in the table. Most of the unit propagation in these formulas was semi-unit propagation.

Domain size	(full-UP / total UP) × domain size	
	NB *moms*	NB *UP*
4	0.84	0.68
8	0.72	0.32
16	0.53	0.16
32	0.50	0.11

Both NB *moms* and the NB *UP* seem to be performing worse, in terms or full-unit propagation, as domain size increases. NB *UP* does especially badly, suggesting that the heuristic is not putting sufficient focus on full-unit clauses. This was a concern I had at the design stage. Pure literal elimination does not occur frequently throughout the search compared to unit propagation, and it occurs less frequently as domain size increases.

The average branching factor can be calculated from the unsatisfiable formulas by dividing the number of backtracks by the number of branches (actually, the formula is $\frac{backtracks-1}{branches} + 1$). This can be done because the search backtracks up all branches for unsatisfiable formulas.

[18] I do not suggest that the UP heuristic finds the *best* variable, but I mean that it does better than the moms heuristic.

Domain size	Average branching factor	
	NB *moms*	NB *UP*
2	2.00	2.00
4	2.51	3.71
8	3.58	6.40
16	4.25	10.87
32	5.31	18.35

It can be seen that the average branching factor is quite a lot lower than the domain size for NB *moms*. This is because some values are eliminated from further inquiry before branching on a variable, either because they do not appear in the formula with that variable any more, or because the a semi-unit propagation has restricted the set of active values for that variable. NB *UP* has a higher average branching factor than NB *moms*, i.e. each branch creates more subtrees on average. Of course, these branches may be better choices (i.e. lead to a solution/contradiction more quickly) than the ones that produce less subtrees, but intuitively a lower branching factor is better. Possibly the branching factor needs more weight in the non-Boolean UP heuristic.

The value of "branches" for the Boolean procedure does not quite fit with what I defined a branch to be; as when it finds a failed literal during a branch (and consequently sets it's value without needing to branch), it still counts it as a branch. Therefore the reported number of branches is more than the number actually carried out. Sadly, I did not have time to repeat the experiments because I noticed this too late.

4.2 Graph colouring

As previously discussed, these formulas represent the minimal colouring and are all satisfiable.

All these formulas are negative, so the AMO clauses are not included in the U and UB translations.

4.2.1 Randomly generated problems

The table entries are marked as "density p (q)", where p is the specified probability of an edge occurring between 2 nodes, and q is the observed density.

Problem	Method	Formula size	Branches	Backtracks	Unit	Pure	Run time (sec)
density 6% (4.3%)	NB *moms*	76	3	0	19	8	0.00
30 nodes	NB *up*		3	0	19	8	0.00
19 edges	U	136	3	0	41	16	0.01
2 colours	UB	196	3	0	63	24	0.01
	B	76	3	0	19	8	0.00
density 23% (24.6%)	NB *moms*	856	10	9	202	8	0.04
30 nodes	NB *up*		6	0	65	3	0.12
107 edges	U	976	13	4	333	23	0.04
4 colours	UB	1216	6	0	147	15	0.04
	B	1712	13	2	26	2	0.07
density 43% (40.2%)	NB *moms*	2100	28	38	778	25	0.16
30 nodes	NB *up*		30	45	963	29	1.56
175 edges	U	2280	11	0	125	34	0.08
6 colours	UB	2760	17	0	176	33	0.11
	B	5600	470	239	1329	211	0.94
density 62% (61.8%)	NB *moms*	4304	52	84	1480	33	0.59
30 nodes	NB *up*		—	—	—	—	>10000.00
269 edges	U	4544	59289	29660	1455344	85059	128.19
8 colours	UB	5264	1207	590	6073	1293	1.35
	B	12912	330	152	264	0	2.28
density 70% (70.6%)	NB *moms*	6140	180	318	4066	112	2.13
30 nodes	NB *up*		—	—	—	—	>10000.00
307 edges	U	6440	—	—	—	—	>10000.00
10 colours	UB	7460	438300	220045	1732979	742125	444.68
	B	20876	—	—	—	—	>10000.00
density 78% (80.5%)	NB *moms*	8400	18	2	249	8	0.81
30 nodes	NB *up*		—	—	—	—	>10000.00
350 edges	U	8760	—	—	—	—	>10000.00
12 colours	UB	10080	34787	17388	114755	33953	26.80
	B	30800	1381042	696708	6278854	280887	4517.04

Up until domain size 6, all the methods are reasonably closely matched. At domain size 8, NB *UP* becomes unfeasible and U starts to show poor performance. At 10 and 12 colours, only UB and NB *moms* have reasonable solution times. Throughout these problems, NB *moms* is by far the best performing method. Unfortunately, I think that the method I used to generate the formulas made those with fewer colours too small; larger problems may have separated the methods for the smaller domain sizes.

These results are particularly surprising, since the Boolean encoding is particularly concise for graph colouring problems — the negative non-Boolean literals are represented as a single Boolean literal and the AMO clauses are not included in the unary and unary-binary translations. I would have expected better performance from the Boolean methods because these are relatively small graph colouring problems. The search tree of the UB method is similar in size to NB *moms* up until domain size 8. It becomes much larger at 10 and 12 colours, again indicating that the heuristic is failing. The result that NB *UP* so suddenly becomes inefficient is also quite intriguing.

For the formulas on which NB *UP* terminated within 3 hours, it produced a similar sized search tree to NB *moms*. This suggests that the moms heuristic does not have such a lower accuracy than the UP heuristic in these formulas, although nothing can be reliably stated about the search tree size of NB *UP* for domain sizes 8, 10 and 12, I would not think that it is worse than that of NB *moms*.

The UB method is the only other method that comes close to the performance of NB *moms*. This is surprising because the UB translation introduces more variables into the translation than the U method. Therefore it cannot be the number of variables alone that stops the U method from operating efficiently.

The binary encoding (B) has good performance until domain sizes 10 and 12, where it suddenly becomes poor. For single value NB variables (e.g $\neg\mathbf{X}\in\{\mathrm{a}_1\}$), the binary translation generally produces formulas with longer clauses than the unary translation (each NB literal is translated as $\lceil \log_2 d \rceil$ Boolean literals). This reduces the amount of unit propagation that can occur early in the Boolean translation because unit clauses take longer to be generated during the search.

The point where most methods suddenly seem to become ineffective is particularly interesting and would, in my opinion, be worth further investigation. Is it due to the increase in domain size (number of colours) or is it because of the density of the graph? Why is NB moms not affected?

I would have liked to have more time to redo these tests with a set of formulas for each domain size, as this would give a better indication of the performance of the various methods. If I were to repeat them, it would be a good idea to fix the search space size (as in the randomly generated NB $(3, \frac{1}{2}d)$-SAT formulas) and repeatedly generate formulas, empirically determining the point where 50% are satisfiable. Unfortunately this was hampered by the poor performance of DP on unsatisfiable graph colouring problems (and the unavailability of a suitable algorithm) and the way the graph generator operated, by interactively requesting various input parameters. I did not have time to re-hash the source code to allow the generator to be systematically called from another program.

4.2.2 Benchmark problems

N-queens

Problem	Method	Formula size	Branches	Backtracks	Unit	Pure	Run time (sec)
n-queens 5×5 5 colours	NB *moms*	3200	6	0	86	0	0.14
	NB *up*		5	0	61	0	1.37
	U	3325	5	0	96	9	0.09
	UB	3625	5	0	117	17	0.09
	B	7680	13	0	25	0	0.31
n-queens 6×6 7 colours	NB *moms*	8120	1521	3278	68043	614	28.36
	NB *up*		22	20	705	3	16.50
	U	8372	30	10	774	46	0.29
	UB	9092	253954	137453	5111813	371860	500.13
	B	23200	2989	1557	7500	229	6.43
n-queens 7×7 7 colours	NB *moms*	13328	11747	29504	688755	1719	505.91
	NB *up*		10	1	292	0	28.07
	U	13671	29	10	1075	37	0.49
	UB	14651	—	—	—	—	>10000.00
	B	38080	2118	1076	7546	43	12.03
n-queens 8×8 9 colours	NB *moms*	26208	—	—	—	—	>10000.00
	NB *up*		—	—	—	—	>10000.00
	U	26784	3594	1816	135641	3548	7.81
	UB	28640	—	—	—	—	>10000.00
	B	84448	—	—	—	—	>10000.00

These formulas show a rather different trend to the random graph problems. Both NB *moms* and NB *UP* have poor performance in terms of run time; NB *moms* has a significantly larger search tree than NB *UP* on 6×6 and 7×7. The UB method is poor on all problems apart from the small one (5×5), B is relatively good but it is U that triumphs over all other methods in terms of run time. The search trees of the NB *UP* and U methods are very similar in size, but on the 6×6 and 7×7 problems, the NB *UP* run time is quite high — averaging 1-3 seconds per branch. On the 8×8 problem, either the lookahead step is too expensive or the heuristic becomes less accurate and the search tree increases in size. The moms heuristic is not preferable to the UP heuristic on these problems — the search tree becomes too large.

These problems have quite a specific structure which may not occur in many other formulas. This may contribute to a particular method being favourable.

Miles graphs

Problem	Method	Formula size	Branches	Backtracks	Unit	Pure	Run time (sec)
miles250	NB *moms*	12384	71	0	294	54	4.49
128 nodes	NB *up*		—	—	—	—	>10000.00
387 edges	U	13408	3181	1552	21288	4148	7.13
8 colours	UB	16480	192	3	536	613	2.03
	B	37152	188	3	60	78	2.29
miles500	NB *moms*	93600	90	0	904	35	68.19
128 nodes	NB *up*		—	—	—	—	>10000.00
1170 edges	U	96160	96	0	786	1674	8.03
20 colours	UB	107424	—	—	—	—	>10000.00
	B	411840	—	—	—	—	>10000.00

The NB *moms* method has a very small search tree for the size of the formula, although they are very slow — 1 branch takes nearly a second for miles500 (remember that this is the moms heuristic, so there is no lookahead). This is because the formula is very large and the setting of individual literal states required during the search takes up time. The UP heuristic cannot hope to match this because the moms heuristic is accurate on this occasion and the extra time needed to perform the lookahead makes NB *UP* take too long.

The unary encoding (U) is the most successful method, NB *moms* seems to do well, but slows on miles500. The UB and B methods perform well on domain size 8 but fall down on domain size 20. Possibly the clauses become too large for B and there are too many variables for UB.

From the methods that terminated within 3 hours, the figures show that there are few backtracks necessary to solve these formulas. They are relatively easy, but their large size creates difficulty.

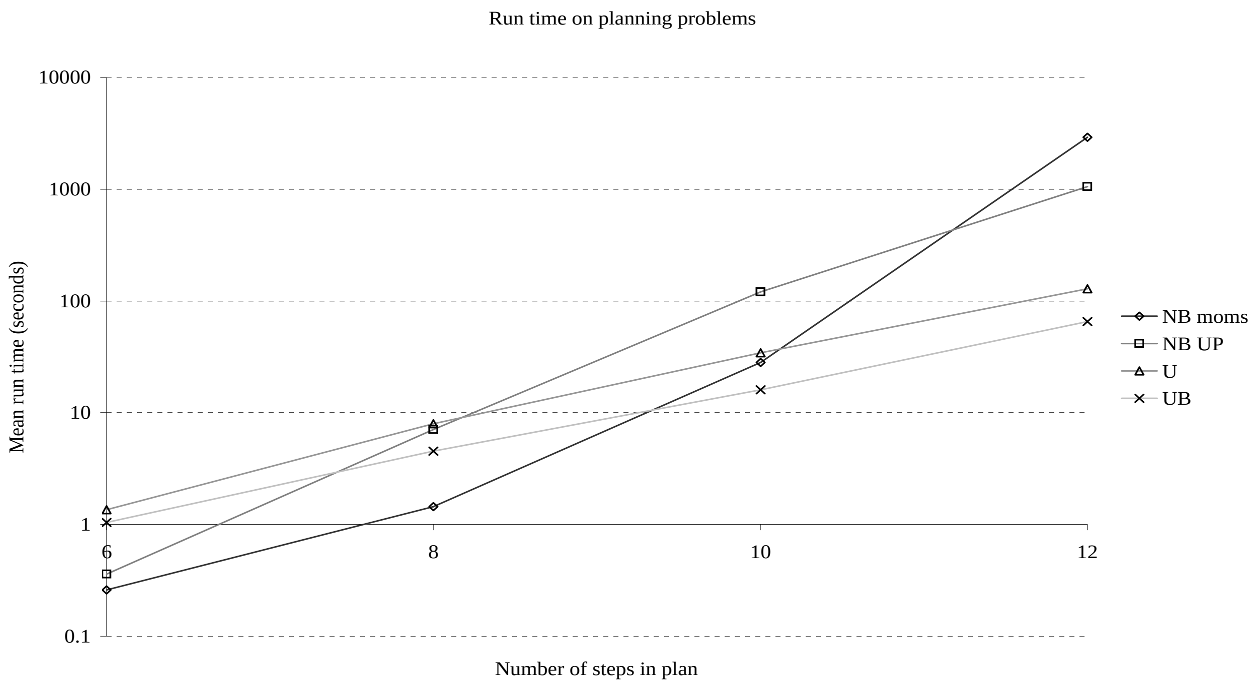
There are only two instances of these problems and they are not entirely representative of real-world problems, so they provide a limited source of information to the evaluation here.

4.3 Planning

These formulas are neither positive or negative in the most concise encoding, so in general both the ALO and AMO clauses are necessary. I tried translating their positive/negative equivalents (thus excluding the ALO/AMO from the translation), but this decreased performance.

All these formulas represent planning problems with the minimal number of steps needed to accomplish the plan (thus they are satisfiable).

Problem	Method	Formula size	Branches	Backtracks	Unit	Pure	Run time (sec)
6 step plan	NB *moms*	3820	2	3	826	0	0.26
	NB *UP*		1	0	692	0	0.36
	U	39532	1	0	1019	0	1.36
	UB	26203	1	0	909	53	1.04
8 step plan	NB *moms*	8940	12	95	6168	0	1.44
	NB *UP*		2	3	2072	0	7.10
	U	165740	6	3	6314	0	7.94
	UB	76364	3	0	3956	94	4.54
10 step plan	NB *moms*	17754	201	2762	146368	0	28.18
	NB *UP*		3	12	5250	0	120.48
	U	528954	14	5	20489	0	34.39
	UB	177930	19	7	10794	122	16.03
12 step plan	NB *moms*	31612	7643	164852	8307612	0	2917.14
	NB *UP*		7	193	32107	0	1051.44
	U	1400476	46	21	93726	0	128.29
	UB	364516	177	88	46775	333	65.49



The trend in the solving methods is the opposite to that seen in the randomly generated NB $(3, \frac{1}{2}d)$-SAT formulas. UB is the best performing method, with a similar growth to U. NB *UP* performs worse on larger formulas than both U and UB, and NB *moms* has the poorest growth. The NB *moms* search tree is very large for the larger domain sizes, as the heuristic becomes less accurate (over 1000 times more branches than the NB *UP* search tree for the 12 step plan) — the inaccuracy of the heuristic is what lets down NB *moms*. NB *UP* is much slower in branching than NB *moms* ($2\frac{1}{2}$ minutes per branch, compared to under $\frac{1}{2}$ second), so it is the lookahead step which is extremely slow. This is what makes NB *UP* slower than the corresponding Boolean translation methods.

The Boolean translations perform well due to the accuracy of the heuristic and the fact that large numbers of non-Boolean literals have a single value (e.g. $\mathbf{X}\in\{\mathrm{a}_1\}$ and $\neg\mathbf{X}\in\{\mathrm{a}_2\}$). These literals are concisely translated and the internal state is easily maintained throughout the search. UB performs better than U, so any negative effect of introducing extra variables is cancelled out by fewer clauses. The only pure literals are those in the UB translation; these are frequent compared to the number of branches in the search tree.

These formulas are large, but solvable with relatively little backtracking by most methods apart from NB *moms*. Having said that, they have relatively little interdependency between actions compared to other planning problems therefore there is less unit propagation fixing the direction of the search. It would be interesting to conduct more experiments on other planning problems, especially unsatisfiable ones which may not follow the same pattern.

Out of all the formulas that I tested, those representing planning problems are the only ones that could utilise variable-specific domains. I conducted tests on these same problems with a modified version of NB-satz which catered for variable-specific domains, but this showed little difference in performance. The clauses that restricted the domains of certain variables are all unit, therefore they are processed before the search begins. There are also relatively few of them (there cannot be more than the number of variables) so the loss in performance from storing them in the data structures is minimal.

The non-Boolean formulas are much smaller than their Boolean translations according to the formula size measure, but the internal data structures of NB-satz are not this much smaller because all literals are stored positively.

I don't think that the power of the heuristic is reduced by the ALO and AMO clauses, because these planning problems are translated including both the ALO and AMO clauses and the best performing methods are the translations — U and UB. Although the accuracy of the heuristic is no doubt reduced with some translations, the extra clauses are not solely responsible.

Conclusion

5.1 Advantages of NB-satz

5.1.1 Semi-unit propagation

Probably the most significant advantage of NB-satz is that it can eliminate certain variable-value pairs from further enquiry (and in turn eliminate variables and clauses from the formula). This fixes the direction of the search where branches would otherwise be. This is a considerably valuable technique since it reduces the number of sub-problems and therefore the amount of work necessary to solve the formula.

This is absent from the Boolean procedure, where the clause **x_a** $\vee$ **x_c** would not be propagated, but the corresponding non-Boolean clause $\mathbf{X}\in\{a,c\}$ would be, and may lead to a significant amount of formula simplification. This factor is the reason why the Boolean procedure performs worse on the formulas which have many literals with large value sets (e.g. the randomly generated formulas); less unit propagation means a larger explored search space.

5.1.2 No extra clauses/variables to enforce non-singular assignments

The ALO and AMO clauses in the unary translation need to be explicitly maintained during the Boolean search. This time overhead is avoided by the non-Boolean procedure, which enforces singular assignments within the algorithm. The extra clauses also make the lookahead step slower.

I originally thought that the heuristic would be affected by the ALO and AMO clauses, but the results from the planning problems suggest otherwise. Those formulas are the largest which I used in testing and therefore have the largest number of ALO and AMO clauses (and many compared to the kernel formula) but the translation methods perform well, suggesting that the heuristic is not affected by the ALO and AMO clauses. Of course, this assumption cannot be proven, since this result was only observed on a number of problems taken from a single class.

5.1.3 The heuristic is not affected by problem reformulation

The non-Boolean UP heuristic generally performs well in terms of search tree size (but not always in terms of time). The search tree of the Boolean Translation is significantly larger than the NB search tree on many formulas including the randomly generated graph colouring problems, suggesting that the heuristic is less accurate. Although the Boolean search does not perform semi-unit propagation, I would suggest that this may not be solely responsible for the poorer performance shown by using the translation methods. If this is so, the Boolean heuristic must in some way be adversely affected by the translation of the non-Boolean formula. Neither of these observations explain the sudden decrease in performance of all other methods except NB *moms* on the randomly generated graph colouring problems — this is something that would be interesting to investigate further.

5.2 Disadvantages of NB-satz

5.2.1 Individual literal states

The non-Boolean unit propagation step in the procedure on page 24 refers to the removal of literals from clauses in the formula. This procedure differs from the Boolean one because a situation can arise where a literal $\mathbf{X}\in\{c,d\}$ is removed, but $\mathbf{X}\in\{a,b\}$ is not. Therefore each *individual* literal must be able to be removed from the formula, independent of all other literals. In the Boolean procedure, all literals x are either LIVE or DEAD — the removal of literals from clauses can be implemented efficiently by setting a single flag for each literal. However the non-Boolean procedure is implemented, it requires a larger amount of state than the Boolean procedure, so the search is slower in comparison.

This problem is created by the semi-unit propagation step, which is critical to the success of the non-Boolean procedure. Unfortunately, the NB procedure cannot have semi-unit propagation without incurring the slow down due to the increased state. Possibly more sophisticated data structures could reduce this problem.

5.2.2 Difficult to find an accurate and fast heuristic

The moms heuristic is often very inaccurate compared to the UP heuristic, in terms of number of branches, as seen on the randomly generated formulas, the n-queens problems and the planning problems. The UP heuristic is accurate, but sometimes extremely slow ($2\frac{1}{2}$ minutes per branch on the largest planning problem). This is a problem and I could not solve it with a restriction for formulas with large domains. Both heuristics are inadequate compared to the performance of the Boolean procedure.

The UP heuristic becomes less easy to restrict as branching commitment (number of subtrees) increases. The more subtrees there are at a single branch, the less different the branches are in terms of average effect of unit propagation at the branch. Chu Min Li [14] shows that a Boolean pure UP heuristic outperforms a Boolean moms heuristic in terms of search time (of course, it also outperforms it in terms of search tree size). The restrictions he presents perform better still, but this trend is not seen on larger domains. On the randomly generated formulas, NB *UP* (non-Boolean pure UP heuristic) outperforms NB *moms* on problems up to and including domain size 4, but at domain size 8, NB *moms* wins. Furthermore, on larger domains, a well-performing restriction of the UP heuristic could not be found. As domain size increases, the extra accuracy of using the UP heuristic is not worth the extra cost.

A solution to this dilemma could be to combat the cause — the number of subtrees at each branch. A UP heuristic may then be more feasible for larger domains and an efficient restriction may be found.

5.3 Comparison to NB-walksat

Non-Boolean Davis-Putnam has quite different benefits over Boolean Davis-Putnam than non-Boolean local search has over Boolean local search. The size of the explored search space is not increased, which is arguably the largest factor in the performance increase found by NB-walksat (in [10]). It is the crucial semi-unit propagation step which allows non-Boolean Davis-Putnam to perform better on formulas containing literals with large value sets. Both NB-satz and NB-walksat benefit from the reduction in problem size resulting from operating on the non-Boolean formula. Both sets of results show similar trends, although the local search procedure (both the Boolean and non-Boolean versions) have significantly shorter search times and can solve larger problems; such is their nature.

5.4 Further work

5.4.1 Different branching strategies

A branch basically refines the current state — a good branch maximises the amount of formula simplification arising from the branch while minimising the commitment. The branching method used by NB-satz carries quite a high commitment at each branch; reducing this commitment is a possible improvement to the search.

With the branching method I used, the non-Boolean search tree is flatter than the Boolean search tree (whichever translation method is used). This is not really a good thing since it puts less importance on the variable order (there is less choice for the heuristic) and more importance on the value order (although not for unsatisfiable formulas). For example: for domain size d and v variables, the flattest tree is of depth 1 and branching factor d^v (this effectively enumerates all solutions if there is no value order). At the other end of the scale, the tallest tree is of depth $(v \log_2 d)$ and branching factor 2. Both have the same search space, but the latter is better because the variable ordering heuristic has more effect on the search. Assuming that a variable ordering heuristic is more accurate than a value ordering heuristic, less of the search space will need to be explored if each node has fewer subtrees.

In addition to this, the more variables there are to choose from, the more choice the heuristic has. More choice for the heuristic suggests that the selected branch will be more likely to be better. I suggest that more choice for the heuristic would make a restricted non-Boolean UP heuristic feasible.

Discussion

Recall that the branching methods that I identified are:

1. For n possible values, each branch produces n subtrees, each making a full commitment to a single value.
2. The set of all values that could be explored is divided into two partitions. Each branch produces 2 subtrees, one making a partial commitment to one partition, the other making a partial commitment to the other partition.
3. For active values V, a branch on variable X and value a is binary, where one subtree makes the full commitment $X \mapsto a$ and the other makes the partial commitment $X \mapsto V \setminus a$.

Both of the alternative methods (2 and 3) make binary branches. For a variable X with n active values, there is 1 possible choice for a branch on X using method 1, n possible choices using method 2 and $S(n, 2)$ possible choices[19] using method 3. There is more choice not only in the variable chosen at each branch, but also in the values instantiated to each subtree. Branching method 2 has as much choice as the unary translation, whereas method 3 has much more, possibly too much to examine all the possibilities.

[19] $S(m, k)$ is the Stirling number representing the number distinct partitions of m distinct elements into k distinct regions.

If a weaker commitment branching strategy is implemented, the UP heuristic will undoubtedly perform differently. Because there will be less commitment at each branch, the resulting amount of formula simplification will be reduced. This is the factor that guides the UP heuristic at present, so it is unclear how it will perform. It will have many more subtrees to examine with branching method 3, so some form of restriction on the partitions that get examined would be necessary. Such a heuristic may be difficult to construct.

Possibly a combination of branching strategies that occur at different depths of the tree may perform well, where the branching factor can be reduced early in the tree and simplification (and commitment) can be increased towards the leaves.

Of course, branching method 2 could be generalised to have n subtrees, with the active values divided between them. This would be trading a higher commitment for potentially more simplification — I am unsure as to what value of n would perform best.

5.4.2 Improvement of the heuristic

The non-Boolean moms heuristic is particularly brutal in it's regard for an unfavourable subtree. Recall that it maximises $\frac{1}{n}\left[min\left(w(X \in \{a_1, \ldots\}), w(X \in \{a_2, \ldots\}), \ldots, w(X \in \{a_n, \ldots\})\right)\right]$, thus it chooses the variable with the best combination of value set size and best "worst subtree". It is the *min* function that selects the worst subtree; this is a straight generalisation of the Boolean heuristic but is perhaps a little out of place in a non-Boolean context. With increasing domain size, there are more subtrees, so basing the variable choice on the worst subtree is a little narrow-minded. I would suggest that a more gentle function would give better performance on problems with larger domains.

So far, the moms heuristic has only considered the literals which have the value a in their value sets. Possibly considering the literals which do *not* have the value a in their value sets may be an important addition to the moms heuristic, since it is these literals which would contribute to future unit propagation. This is the "remove clauses/remove literals" debate from the value order section all over again.

The UP heuristic seems to need to be more focused on literals with smaller value set sizes, to minimise the branching factor. This can be seen from the relative performance of NB *UP* and NB *moms* on the randomly generated formulas. A different branching strategy may allow a more efficient UP heuristic, due to more choice for the heuristic, as discussed above.

5.4.3 Dependency-directed backtracking

Dependency-directed backtracking can significantly increase the performance of Boolean DP on some problems [17]. I conjecture that a similar performance increase can be gained for non-Boolean DP.

5.4.4 Randomisation and restarts

The heuristic may identify more than one "best" branch. Satz chooses the last of these candidates, but a random factor can be introduced without compromising the heuristic by randomly selecting one of these candidates as the branching variable. This random tie breaking is effective in Boolean DP, in combination with restarts[20]. I think that non-Boolean DP would show the same benefit.

[20] Satz-rand, a randomised version of satz is available from SATLIB.

5.4.5 Modification of a Boolean solver

A different approach to solving non-Boolean problems is to modify a Boolean solver to tackle some of it's shortcomings on solving translations of non-Boolean formulas. I do not believe that this approach has been taken before, but I suggest that it could be fruitful. For instance, the Boolean heuristic could be modified to only consider kernel clauses, thus it would not at all be affected by the ALO and AMO clauses and the extra variables in the unary-binary translation would not be branching candidates. A non-CNF Boolean DP solver could be used to work with binary translations, as these are not concisely expressed in CNF.

5.4.6 A non-CNF solver

Certain problems get very large when translated to CNF, and it is this blow-up, rather than the actual problem that makes solving difficult. The CNF translation step could be avoided (much like the Boolean translation is avoided with a non-Boolean solver) by creating a non-CNF solver.

5.4.7 A special-purpose graph colouring algorithm

The non-Boolean Davis-Putnam procedure takes part of the problem specification (the constraint of singular assignments) and moves this into the *algorithm* rather than the input data. This same technique can be applied to graph colouring problems, by placing the constraint that all adjacent nodes are different colours in the algorithm rather than the problem encoding, as it would be if it was encoded in SAT. A benefit arising from this is that the state is reduced, which means a quicker search. Also, the size of the problem encoding will not be dependent on the number of colours.

5.5 Summary

It has been shown that NB-satz outperforms satz on a number of non-Boolean problems, despite using a heuristic which is either less powerful or slower to compute. It does performs worse on some classes of problem, but the Boolean procedure has a better heuristic and allows optimisations that reduce the complexity of certain steps in the search. I have identified points where the non-Boolean procedure has particular benefits over the translation approach and I have determined points where it is at a disadvantage. I speculate that the performance of the non-Boolean procedure can be increased by various developments, such as different branching methods, an improved heuristic and the techniques of dependency-directed backtracking and randomisation and restarts.

In developing a non-Boolean Davis-Putnam procedure, I have shown that this approach is not equivalent to the translation approach. The results show large differences between the two methods. This study has suggested new directions that can be explored to further develop the non-Boolean procedure.

The code for NB-satz is available on the Internet at:
`http://www-users.york.ac.uk/~pgs103/sat.html`

Acknowledgments

I thank Alan Frisch, my supervisor, for his guidance throughout this project. Thank you to my parents and ♡Viji for proofreading my report and to Shirley for her enthusiasm. Thanks to Kerry, who helped me with the finishing touches.

Bibliography

[1] P. Cheeseman, B. Kanefsky and W. M. Taylor, “Where the really hard problems are”, Proceedings of the IJCAI-91: pages 163–169, 1999.

[2] S. A. Cook, “The complexity of theorem-proving procedures”, Proceedings of the Annual ACM Symposium on Theory of Computing 151–158, 1971.

[3] J. M. Crawford. “Solving satisfiability problems using a combination of systematic and local search”, Second DIMACS Challenge: Cliques, Coloring, and Satisfiability, Rutgers University, NJ., 1993.

[4] J. M. Crawford and L. D. Auton, “Experimental results on the crossover point in random 3SAT”, Artificial Intelligence, vol. 81, 1996.

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

[6] M. Davis, G. Logemann and D. Loveland, “A machine program for theorem proving”, CACM 5:394–397, 1969.

[7] O. Dubois, Y. Boufkhad et al., ”SAT versus UNSAT, second DIMACS challenge”, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (AMS), 415-436, 1996.

[8] J. Franco and M. Paull, “Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem”, Discrete Applied Mathematics 5(1):77–87, 1983.

[9] J. Freeman, “Improvements to propositional satisfiability search algorithms” , PhD thesis, The University of Pennsylvania, 1995.

[10] A. M. Frisch and T. J. Peugniez, “Solving non-Boolean problems with stochastic local search: a comparison of encodings”, 1999.

[11] A. T. Goldberg, “On the complexity of the satisfiability problem”, Courant Computer Science Report No. NSO-16, Courant Institute of Mathematical Sciences, Computer Science Department, New York University, 1979.

[12] J. N. Hooker and V. Vinay, “Branching rules for satisfiability”, Journal of Automated Reasoning 15:359–383, 1995.

[13] R. G. Jeroslow and J. Wang, “Solving propositional satisfiability problems”, Annals of Mathematics and Artificial Intelligence 1:167–187, 1990.

[14] C. M. Li and Anbulagan, “Heuristics based on unit propagation for satisfiability problems”, Proceedings of the 15th IJCAI: pages 366–371, International Joint Conference on Artificial Intelligence, 1997.

[15] T. J. Peugniez, “Solving non-Boolean satisfiability problems with local search” BSc Dissertation, Department of Computer Science, University of York, March 1998.

[16] H. Zhang, M. Stickel, “Implementing Davis-Putnam's method by tries”, Technical Report, University of Iowa, 1994.

[17] H. Zhang, “SATO: an efficient propositional prover”, Proceedings of the International Conference on Automated Deduction 1997.