

Algorithms for SATISFIABILITY

Course notes for Search and Optimization

Spring 2004

Peter Bro Miltersen

April 19, 2004

Version 4.0

1 Basic issues

SATISFIABILITY is the following decision problem: Given a Boolean formula in conjunctive normal form, return “yes” if the formula has a satisfying assignment and “no” otherwise. SATISFIABILITY plays a crucial role in the theory of **NP**-completeness. Also, algorithms solving SATISFIABILITY are important in practice and are routinely used, for instance, for hardware verification. Thus, SATISFIABILITY is important from a theoretical as well as from a practical perspective. In these notes, we discuss algorithms for SATISFIABILITY.

By Cook’s theorem, SATISFIABILITY is **NP**-complete. Thus, under the assumption $\mathbf{P} \neq \mathbf{NP}$, SATISFIABILITY does not have a polynomial algorithm. By making a somewhat stronger, but still very reasonable assumption, a stronger conclusion can be made.

Definition The complexity class **SUBEXP** (read “subexponential time”) is the class of languages L for which there, for every real constant $\epsilon > 0$ is a deterministic Turing machine M and a positive integer N so that M decides L and so that M takes at most $2^{|x|^\epsilon}$ steps on any input x of length $|x| > N$.

Like **P**, **SUBEXP** is a *robust* class, i.e., we define it using the clean and convenient Turing machine model, but we would define the same class if we used any other “reasonable” sequential computational model, such as a random access machine.

To simulate all of **NP**, we expect exponential time to be necessary. Thus, it is widely conjectured that **NP** is not a subset of **SUBEXP**. This makes the following Proposition interesting.

Proposition 1 *If $\mathbf{NP} \not\subseteq \mathbf{SUBEXP}$, then for every **NP**-complete language $L \subseteq \Sigma^*$, there is a constant $\epsilon > 0$ so that for any machine M that decides L , M takes at least $2^{|x|^\epsilon}$ steps on input x , for infinitely many $x \in \Sigma^*$.*

Proof Exercise 1.2.

Thus, for any **NP**-complete language and in particular SATISFIABILITY, we do not expect an algorithm running in time $2^{|x|^{\epsilon(1)}}$. What we are aiming for are algorithms with as “mild” an exponential behavior as possible.

When analyzing concrete algorithms for SATISFIABILITY, we do not usually express the running time in terms of the length of an encoding of the instances over a finite alphabet as above. It is more relevant for the applications to express the complexity in terms of m , the number of *clauses* of

the formula or n , the number of *variables* of the formula. It is, however, convenient to keep the length l of the encoding of the formula around as a secondary parameter.

When designing and analyzing algorithms for SATISFIABILITY we are going to focus on only one of the parameters n or m . If we focus on n , we are going to express the complexity of the algorithm as an expression of the form

$$T(n, l) \leq 2^{f(n)} l^{O(1)},$$

while, if we focus on m , we are going to express the complexity as an expression of the form

$$T(m, l) \leq 2^{g(m)} l^{O(1)}.$$

Clearly, we want an algorithm behaving well as a function of n if there are few variables compared to the number of clauses in the instance(s) we plan to solve. Similarly, we want an algorithm behaving well as a function of m if the opposite is the case. In either case, the factor $l^{O(1)}$ means that we only care about the complexity up to a polynomial in the length of the formula, i.e., polynomial processing of the formula at any point in time is “for free”. This makes our time bounds robust, i.e., they are valid for any reasonable sequential model of computation, such as the Turing machine or the more practical random access machine. This is convenient as long as we discuss theoretically efficient algorithms. When it comes to implementation, we do of course care about minimizing the polynomial overhead in l and do so by carefully selecting and using appropriate data structures for representing formulas and truth assignments. Such practical issues are deferred to Section 4.

As an example, consider the trivial algorithm for SATISFIABILITY consisting in trying all possible truth assignments to the variables of the formula. Clearly, the complexity of this algorithm can be expressed as

$$T(n, l) \leq 2^n l^{O(1)}.$$

Somewhat surprisingly, this is the very best known upper bound for SATISFIABILITY in terms of n unless we restrict the length of the clauses! Obtaining better bounds under such restrictions is the topic of Section 3.

First, however, we shall deal with an algorithm (or, more accurately, a class of algorithms) for SATISFIABILITY whose complexity can also be described

```

boolean DavisPutnam(CNFformula  $f$ )
   $f := \text{Simplify}(f)$ ;
  if  $f$  does not contain any variables then
    return Evaluate( $f$ )
  else
    Pick a variable  $x$  in  $f$ ;
     $f_0 := f[x \leftarrow \text{false}]$ ;
    if DavisPutnam( $f_0$ )=true then
      return true
    else
       $f_1 := f[x \leftarrow \text{true}]$ ;
      return DavisPutnam( $f_1$ )
  fi
fi

```

Figure 1: The Davis-Putnam procedure.

as $T(n, l) \leq 2^n l^{O(1)}$ if we focus on the parameter n , but if we instead express the complexity as

$$T(m, l) \leq 2^{g(m)} l^{O(1)},$$

we can get a non-trivial bound $g(m) = \alpha m$ for various constants $0 < \alpha < 1$ depending on how we specify the details of the algorithm. The algorithm is the celebrated *Davis-Putnam* procedure, first described in 1962, though the variation giving the best known value of α is from 1998.

Exercise 1.1 Show that $\mathbf{P} \subseteq \mathbf{SUBEXP}$.

Exercise 1.2 Prove Proposition 1.

2 The Davis-Putnam procedure

The Davis-Putnam procedure originates in Davis, Logemann and Loveland (1962). Its name is something of a mistake or misunderstanding - the paper by Davis and Putnam (1960) actually describes a very different procedure for solving satisfiability! The name is, however, well-established, so we shall make no attempt to correct this mistake. An outline of the Davis-Putnam procedure can be seen in Figure 1. In the description of the algorithm we

allow a CNF formula to contain the constants **true** and **false** in addition to Boolean variables and their negations. The variables, the negations of variables, and the constants of the formula are called the *literals* of the formula. We shall usually denote the number of literals in the formula given as input by r .

Note that a formula containing no variables but only constants can be readily evaluated to either **true** or **false** by doing the necessary Boolean operations; this is the intended semantics of the expression “Evaluate(f)”. By $f[x \leftarrow \mathbf{true}]$ we mean the formula obtained by replacing all occurrences of the literal x with **true** and all occurrences of the literal $\neg x$ with **false**. We define $f[x \leftarrow \mathbf{false}]$ similarly. If we ignore the first line $f := \text{Simplify}(f)$ for a moment, we see from Figure 1 that the Davis-Putnam procedure decides if a formula is satisfiable by first recursively deciding if it is satisfiable by an assignment making a particular variable false, and if not, recursively deciding if it is satisfiable by an assignment making the same variable true. Thus, at first sight, we have a simple uncontrolled exponential branching algorithm, clearly correct, and clearly of complexity $2^n l^{O(1)}$.

The key to obtaining a non-trivial version of the Davis-Putnam procedure is the line $f := \text{Simplify}(f)$. Simplify is intended to be an auxiliary method in our program. Intuitively, we want $\text{Simplify}(f)$ to return a “simplified” version of f with fewer clauses or variables, so that our job is made easier. Actually, to preserve partial correctness of the procedure, the only property of the method we shall need is the following.

Crucial property of Simplify If g is the Boolean formula that $\text{Simplify}(f)$ returns, then g is satisfiable if and only if f is satisfiable.

The variations of the Davis-Putnam procedure that we shall examine are essentially obtained by designing various Simplify methods obeying the Crucial property.

2.1 The basic Simplify method

The Simplify methods we shall design are all conveniently described by a set of *rewrite rules*, describing how the Simplify method may turn the formula into a simpler one. The first five rewrite rule we shall consider are essentially those originally considered by Davis, Logemann and Loveland.

Rule I. If a clause contains a variable in both unnegated and negated form, replace the two variables by the constant **true**. If a clause contains two occurrences of the same variable in the same form, replace the two occurrences

```

CNFformula SimplifyS(CNFformula f)
  while some rule of S applies to f do
    apply the first applicable rule of S to f
  od
return f

```

Figure 2: Auxiliary method Simplify.

by one. If the set of literals in one clause C_1 is a subset of the literals in another clause C_2 , remove C_2 .

Rule II. If a clause contains the constant **true**, remove the clause from the formula. If the clause was the only clause in the formula, replace the formula by **true**.

Rule III. If a clause contains the constant **false**, remove the constant from the clause. If this causes the clause to be empty, replace the entire formula by **false**.

Rule IV. If a clause only contains a single unnegated variable x , replace every occurrence of x in f with **true**. If a clause contains only a single negated variable $\neg x$, replace every occurrence of x in f with **false**.

Rule V. If the formula contains some variable in only unnegated form (e.g., the formula contains x but not $\neg x$), replace the variable by **true** everywhere in the formula. Similarly, if the formula contains some variable in only negated form, replace the variable by **false** everywhere in the formula.

With these five rewrite rules, we get our first version of the Simplify method. In general, our Simplify method will follow the pattern of Figure 2, where S is a list of rewrite rules. Thus, our first version of the Simplify method is $\text{Simplify}_{\text{I,II,III,IV,V}}$ or $\text{Simplify}_{\text{I-V}}$ for short.

It is easy to see that $\text{Simplify}_{\text{I-V}}$ satisfies the Crucial Property of Simplify. Giving a proof is the topic of Exercise 2.1. Furthermore, we note that $\text{Simplify}_{\text{I-V}}$ has complexity polynomial in l , the number of symbols in an encoding of f , because

1. it can be checked if a rule applies to f in time polynomial in l ,
2. applying a rule can be done in time polynomial in l ,
3. each application of one of Rules I-III reduces the number of literals in the formula without increasing the number of variables in the formula

while each application of one of Rules IV-V reduces the number of variables in the formula without increasing the number of literals. Thus, if the formula originally contained n variables and r literals, we will go through at most $n + r \leq 2l$ iterations of the loop.

Furthermore, we have the following Proposition:

Proposition 2 *The Davis-Putnam procedure using Simplify_{I-V} runs in time at most $2^m l^{O(1)}$*

Proof Rules I-IV ensure that after simplification, each formula on which the procedure is recursively called contains at most $m - 1$ clauses, if the original formula contained m . Indeed, Rule IV ensures that the variable on which we are branching occurs both negated or only unnegated in the formula and Rule II thus ensures that a clause is removed in each recursive call. Furthermore the total length of the encoding of each new formula is smaller than the length of the original formula. Thus, if we let $T(m, l)$ denote an upper bound on the worst case time for running Davis-Putnam on a formula containing at most m clauses and with an encoding of size at most l , we get the recurrence $T(m, l) = T(m - 1, l) + T(m - 1, l) + l^{O(1)}$ for $m \geq 1$ and $T(0, l) = l^{O(1)}$, with the solution $T(m, l) = 2^m l^{O(1)}$. **QED**

2.2 Adding Resolution

To improve the asymptotic running time of the algorithm as a function of m , we need to make sure that more than one clause is removed in the simplified versions of the formulas on which we call Davis-Putnam recursively.

To ensure this, we need a rewrite rule based on *resolution*.

Definition (Resolving a variable in a formula) Let a CNF formula f be given and let x be some variable x in f . Let C_1, C_2, \dots, C_p be the clauses in f where x occurs unnegated, but with x removed and D_1, D_2, \dots, D_q be the clauses in f where x occurs negated, but with $\neg x$ removed. That is, f is the formula

$$\begin{aligned} & (x \vee C_1) \wedge (x \vee C_2) \cdots (x \vee C_p) \\ \wedge & (\neg x \vee D_1) \wedge (\neg x \vee D_2) \cdots (\neg x \vee D_q) \\ \wedge & U, \end{aligned}$$

where U is the conjunction of the clauses *not* containing x or $\neg x$.

Then, $\text{Resolve}(f, x)$ is defined to be the CNF formula

$$\begin{aligned} & (C_1 \vee D_1) \wedge (C_1 \vee D_2) \wedge \cdots \wedge (C_1 \vee D_q) \\ \wedge & (C_2 \vee D_1) \wedge (C_2 \vee D_2) \wedge \cdots \wedge (C_2 \vee D_q) \\ \wedge & \cdots \\ \wedge & (C_p \vee D_1) \wedge (C_p \vee D_2) \wedge \cdots \wedge (C_p \vee D_q) \\ \wedge & U. \end{aligned}$$

Thus, if f contains m clauses, $\text{Resolve}(f, x)$ contains $m - (p + q) + pq$ clauses and does not contain the variable x .

Lemma 3 (Resolution Lemma) *f is satisfiable if and only if $\text{Resolve}(f, x)$ is satisfiable.*

Proof Exercise 2.2.

We can now add a sixth rewrite rule, the *resolution rule*.

Rule VI. If the formula f contains some variable x occurring p times negated and q times unnegated, or vice versa, for integers p, q so that $pq \leq p + q$, replace f by $\text{Resolve}(f, x)$. If there is more than one such variable, choose one with $pq < p + q$ over one with $pq = p + q$.

Our next version of Davis-Putnam uses $\text{Simplify}_{\text{I,II,III,IV,V,VI}}$, or, for short, $\text{Simplify}_{\text{I-VI}}$. The correctness is immediate from the proof of the correctness of $\text{Simplify}_{\text{I-V}}$ combined with the Resolution Lemma. For the complexity, we should establish that $\text{Simplify}_{\text{I-VI}}$ runs in time $l^{O(1)}$ on formulas of encoding length l . This is not quite obvious, as rewrite Rule VI, in contrast to the other rewrite rules may actually make the number of literals in the formula *increase*. Note, however, that it makes the number of variables strictly *decrease* without increasing the number of clauses, and that the number of literals in any clause is at most $O(n)$, because of Rule I. Thus, on a formula containing n variables, Rule VI will be used at most n times, and each time the total number of literals in the resulting formula is at most $O(mn)$. All other rules strictly decrease the total number of literals in the formula or the number of variables in the formula without increasing the number of variables, the number of clauses, or the number of literals. Thus, we will go through at most $O(n^2m)$ iterations of the while loop and in each step only handle a formula with at most $O(mn)$ literals. As $m, n \leq l$, we conclude that $\text{Simplify}_{\text{I-VI}}$ has polynomial complexity in l .

After having applied $\text{Simplify}_{\neg V_1}$ to f we can conclude that for every variable occurring in the formula, if we let p be the number of times it occurs positively and q be the number of times it occurs negatively then $p + q < pq$. This implies that $\min(p, q) \geq 2$ and $\max(p, q) \geq 3$.

Note that if a variable x occurs p times positively in the simplified formula f with m clauses, then the number of clauses in $\text{Simplify}(f[x \leftarrow \mathbf{true}])$ is at most $m - p$, because of Rule II. Similarly, if $\neg x$ occurs q times in f , then the number of clauses in $\text{Simplify}(f[x \leftarrow \mathbf{false}])$ is at most $m - q$.

This means that if we let $T(m, n)$ denote an upper bound on the worst case time for running Davis-Putnam using $\text{Simplify}_{\neg V_1}$ on a formula containing at most m clauses and n variables, we get the recurrence

$$\begin{aligned} T(m, n) &= T(m - 2, n) + T(m - 3, n) + (mn)^{O(1)} \text{ for } m \geq 3, \\ T(m, n) &= (mn)^{O(1)} \text{ for } m \leq 2. \end{aligned}$$

This recurrence is not so readily solved as the one in the proof of Proposition 2. To solve it, we shall prove a general theorem which is very useful for solving recurrences of this type (i.e., *linear recurrences*).

Theorem 4 *Let k be a positive integer and let $f : \mathbf{N} \rightarrow \mathbf{R}$ satisfy*

- $f(n) \leq a_1 f(n - 1) + a_2 f(n - 2) + \dots + a_k f(n - k)$ for $n \geq k$,

where $a_1, a_2, \dots, a_k \in \mathbf{R}^+$. Then $f(n) = O(v^n)$ where v is the unique positive real root of

$$g(t) = 1 - \frac{a_1}{t} - \frac{a_2}{t^2} - \dots - \frac{a_k}{t^k}.$$

Proof The function g is monotonely increasing, $g(t) \rightarrow -\infty$ as $t \rightarrow 0+$, and $g(t) \rightarrow 1$ as $t \rightarrow +\infty$. Thus g does indeed have a unique positive real root and v is thus well-defined.

We now show by induction in n that $f(n) \leq cv^n$ for some constant c and all $n \geq 0$. For the basis of the induction, we simply pick c so large that the claim is true for $0 \leq n \leq k$.

Now let $n > k$ and suppose that $f(i) \leq cv^i$ for all integers i with $0 \leq i < n$.

	1	2	3	4	5	6	7	8	9	10
1	100000	69425	55147	46496	40569	36200	32818	30107	27876	26002
2	69425	50000	40569	34713	30627	27574	25183	23248	21643	20285
3	55147	40569	33334	28777	25564	23142	21233	19679	18383	17281
4	46496	34713	28777	25000	22319	20285	18674	17357	16254	15314
5	40569	30627	25564	22319	20000	18235	16830	15679	14712	13885
6	36200	27574	23142	20285	18235	16667	15417	14389	13523	12782
7	32818	25183	21233	18674	16830	15417	14286	13354	12569	11895
8	30107	23248	19679	17357	15679	14389	13354	12500	11779	11160
9	27876	21643	18383	16254	14712	13523	12569	11779	11112	10537
10	26002	20285	17281	15314	13885	12782	11895	11160	10537	10000

Figure 3: $\lceil (\log_2 v_{i,j})10^5 \rceil$ with $v_{i,j}$ being the unique real root of $1 - 1/t^i - 1/t^j$.

Then,

$$\begin{aligned}
f(n) &= a_1 f(n-1) + a_2 f(n-2) + \dots + a_k f(n-k) \\
&\leq a_1 c v^{n-1} + a_2 c v^{n-2} + \dots + a_k c v^{n-k} \\
&= c v^n (a_1 v^{-1} + a_2 v^{-2} + \dots + a_k v^{-k}) \\
&= c v^n (1 - g(v)) \\
&= c v^n,
\end{aligned}$$

as was to be shown.

When using Theorem 4, the table of Figure 3 is often useful.

Theorem 5 *Davis-Putnam using Simplify_{I-VI} solves SATISFIABILITY in worst case time at most $2^{0.40569m} l^{O(1)}$.*

Proof By the discussion above, the running time of the procedure can be bounded by the solution to the recurrence $T(m, n) = T(m-2, n) + T(m-3, n) + p(mn)$ for $m \geq 0$ and $T(m, n) = p(mn)$ for $m \leq 0$ where p is some polynomial. If we define $g(m, n) = (T(m, n)/p(mn) + 1)/2$, we have that $g(m, n) = g(m-2, n) + g(m-3, n)$ for $m \geq 3$ and $g(m, n) \leq 1$ for $m \leq 2$. Thus, the solution f to $f(m) = f(m-2) + f(m-3)$ for $m \geq 1$ and $f(m) = 1$ for $m \leq 0$ is an upper bound for $g(m, n)$. By Theorem 4, we have that $g(m, n)$ is $O(v^m)$ where v is the real root of $t \rightarrow 1 - 1/t^2 - 1/t^3$. According to the table in Figure 3, 0.40569 is an upper bound on $\log v$. Therefore $T(m, l) = (2g(m) - 1)p(mn) \leq 2^{0.40569m} (mn)^{O(1)} \leq 2^{0.40569m} l^{O(1)}$. **QED**

2.3 The Monien-Speckenmeyer algorithm

The next simplification rule we add to improve the asymptotic performance of the Davis-Putnam procedure was given by Monien and Speckenmeyer in 1980.

Rule VII. If every variable in f has the property that it occurs twice unnegated and at least three times negated or vice versa, let a be the truth assignment which assigns **true** to a variable if and only if it occurs more often unnegated than negated in f . If a satisfies f , replace f by **true**, otherwise leave f unchanged.

Clearly, Rule VII obeys the Crucial Property of Simplify: If we can actually exhibit a satisfying assignment to f , we are allowed to replace f by **true**. Also, adding Rule VII clearly does not blow up the complexity of $\text{Simplify}_{\text{I-VI}}$ significantly; $\text{Simplify}_{\text{I-VII}}$ also runs in time $l^{O(1)}$.

We have to augment the Davis-Putnam procedure itself slightly to get the desired effect of adding Rule VII. To be precise, we have to specify how we pick the variable x on which to branch. We say that x is *allowable* for f if one of the following cases are true:

- x appears at least three times in f and so does its negation $\neg x$.
- x appears two times in f and its negation appears at least three times in f . Furthermore, for one of the clauses C where x occurs the following holds: *all* literals in the clause appear exactly twice in f in the form that they appear in C and at least three times in the negated form.
- $\neg x$ appears two times in f and x appears at least three times in f . Furthermore, for one of the clauses C where $\neg x$ occurs the following holds: *all* literals in the clause appear exactly twice in f in the form that they appear in C and at least three times in the negated form.

We need the following lemma.

Lemma 6 *If f is the output of $\text{Simplify}_{\text{I-VII}}$ and f contains some variable then it contains an allowable variable.*

Proof As argued previously, Rules I-VI ensure that every variable will occur at least twice unnegated and at least three times negated or vice versa. If some variable appears at least three times negated and three times unnegated it is allowable. Thus, we can assume that each variable appears exactly twice negated and at least three times unnegated or vice versa. Now, Rule VII

```

boolean MonienSpeckenmeyer(CNFformula  $f$ )
   $f := \text{Simplify}_{\text{I-VII}}(f)$ ;
  if  $f$  does not contain any variables then
    return evaluate( $f$ )
  else
    Pick an variable  $x$  allowable for  $f$ ;
     $f_0 := f[x \leftarrow \text{false}]$ ;
    if MonienSpeckenmeyer( $f_0$ )=true then
      return true
    else
       $f_1 := f[x \leftarrow \text{true}]$ ;
      return MonienSpeckenmeyer( $f_1$ )
  fi
fi

```

Figure 4: The Monien-Speckenmeyer algorithm.

ensures that if each clause contains a literal (i.e., a variable or its negation) that appears three times, then the entire formula is replaced by **true**, since the assignment a satisfies such a formula. Thus, there must be some clause containing only literals appearing twice. Any variable in such a clause is allowable. **QED**

Lemma 6 ensures that the Monien-Speckenmeyer algorithm of Figure 4 is well-defined.

Theorem 7 *The Monien-Speckenmeyer algorithm solves SATISFIABILITY in time $2^{m/3}l^{O(1)}$.*

Proof We shall argue that the running time of the Monien-Speckenmeyer algorithm satisfies the recurrence $T(m, n) \leq T(m - 3, n) + T(m - 3, n) + (mn)^{O(1)}$. This immediately yields the statement of the theorem.

The procedure branches according to some allowable variable x in f . If x and its negation each appears at least three times in f , each of the formulas on which we apply the procedure recursively will contain at most $m - 3$ clauses, hence satisfying the desired recurrence.

Otherwise either x appears twice in f and $\neg x$ appears at least three times in f or vice versa. Assume without loss of generality that the first case applies. Then, after simplification, the formula f_0 contains at most $m - 3$ clauses. At first sight, the simplified formula f_1 may contain as many as $m - 2$ clauses.

We shall argue that it actually also contains at most $m - 3$ clauses and we will be done.

Since x is allowable, it is contained in a clause C containing literals only appearing twice. It is also contained in another clause C' . Because of Rule IV, C contains at least one other literal besides x . When x is replaced by **true**, Rule II will ensure the clause is removed. Thus, each of the other literals now only appear in one other clause. Because of Rule I, one of these clauses, say, C'' , is different from C' . After simplification, C and C' have been removed from f_1 and Rule VI has merged C'' with another clause. Thus we get rid of at least three clauses and are done. **QED**

2.4 The algorithm of Hirsch

The worst-case time bound of the Monien-Speckenmeyer algorithm is very close to the best known bound. Inspecting our progress so far, it seems that the next thing we could hope for would be an algorithm whose running time is bounded by the solution to the recurrence

$$T(m, n) = T(m - 3, n) + T(m - 4, n) + (mn)^{O(1)},$$

yielding a running time of $2^{0.28777m}l^{O(1)}$. It is, however, not known at the time of writing if this is possible. Thus, obtaining such an algorithm is an open problem.

The best known bound is obtained by an algorithm of Hirsch (1998), with a running time bounded by the solution to the recurrence

$$T(m, n) = 2T(m - 6, n) + 2T(m - 7, n) + (mn)^{O(1)},$$

yielding a running time of $2^{0.30897m}l^{O(1)}$.

Though the algorithm of Hirsch does follow the Davis-Putnam pattern, the details of the algorithm and its analysis are technical and shall not be explained here. Let us merely mention that an informal explanation of the recurrence is as follows: The algorithm of Hirsch may occasionally make two recursive calls on two formulas only three clauses shorter than the present one, but the analysis of the algorithm then guarantees that in each of the two resulting branches, the *next* two recursive calls will have the property that at least 4 clauses will be removed in the formula of one of the branches and at least 3 clauses will be removed in both formulas. Thus, looking “two steps ahead”, we see that after having branched on two variables, we will have four formulas to consider, two containing at least seven fewer clauses than

the original formula and two containing at least six fewer clauses, yielding the recurrence above.

2.5 Exercises

Exercise 2.1 Show that Simplify_{I-V} satisfies the Crucial Property of Simplify.

Exercise 2.2 Prove Lemma 3.

Exercise 2.3 Which is asymptotically faster in the worst case, an algorithm whose time bound obeys the recurrence $T(n) = T(n-1) + T(n-10)$ or one whose time bound obeys the recurrence $T(n) = T(n-3) + T(n-5)$?

Exercise 2.4 Does any of the given variations of the Davis-Putnam procedure solve the 2SAT problem in polynomial time? Can you make a variation that does?

Exercise 2.5 Show that the Monien-Speckenmeyer algorithm solves the SAT-ISFIABILITY problem in time $2^{r/6}l^{O(1)}$, where r is the number of *literals* in the CNF formula given as input and l is the encoding length of the formula. Can you make an algorithm doing even better as a function of r ? The world record, due to Hirsch (1998), is $2^{0.10299r}l^{O(1)}$

Exercise 2.6 The Davis-Putnam procedure in all the variations given solves the *decision* version of the SATISFIABILITY problem, i.e., it returns a Boolean. Show how to modify the procedure(s) to also return a satisfying assignment in case there is one.

Exercise 2.7 The Davis-Putnam procedure, as stated, works only for CNF formulas, as this is the kind of formulas the Simplify method, as stated, deals with. Make a variation of the procedure that works for *arbitrary* Boolean formulas. Express its complexity as $2^{\alpha r}l^{O(1)}$ where l is the encoding length of the formula and r is the number of literals in the formula (i.e., the number of leaves in the formula if we view it as a tree). How small can you make α ?

Exercise 2.8 MAX INDEPENDENT SET is the problem of finding the biggest independent set of vertices in a graph. We measure the complexity

of algorithms solving the problem as a function of the number of vertices, n , in the graph.

a) Show that an obvious algorithm for MAX INDEPENDENT SET has a running time of $2^n n^{O(1)}$.

The maximum degree of a graph is the maximum number of edges adjacent to any particular vertex.

b) Show that the MAX INDEPENDENT SET problem can be solved in polynomial time for graphs of maximum degree 0. Derive from this an algorithm for MAX INDEPENDENT SET obeying the recurrence relation $T(n) \leq T(n-1) + T(n-2) + n^{O(1)}$. Give an upper bound for the complexity of the algorithm in the form $O(2^{\alpha n})$ for a constant $\alpha < 1$.

c) Show that the MAX INDEPENDENT SET problem can be solved in polynomial time for graphs of maximum degree at most 1. Derive from this an algorithm for MAX INDEPENDENT SET obeying the recurrence relation $T(n) \leq T(n-1) + T(n-3) + n^{O(1)}$. Give an upper bound for the complexity of the algorithm in the form $O(2^{\beta n})$ for a constant $\beta < 1$.

d) Show that the MAX INDEPENDENT SET problem can be solved in polynomial time for graphs of maximum degree at most 2. Derive from this an algorithm for MAX INDEPENDENT SET obeying the recurrence relation $T(n) \leq T(n-1) + T(n-4) + n^{O(1)}$. Give an upper bound for the complexity of the algorithm in the form $O(2^{\gamma n})$ for a constant $\gamma < 1$.

e) Can you obtain an even better algorithm for MAX INDEPENDENT SET?

Exercise 2.9 Show that the “obvious” algorithm for k -COLORING has complexity $k^n n^{O(1)}$. Design an algorithm for k -COLORING running in time $(k-1)^n n^{O(1)}$. **A more difficult challenge:** Can you design an algorithm for 3-COLORING running in time $O(2^{\alpha n})$ for a constant $\alpha < 1$?

3 Covering code based algorithms

In this section, we describe some algorithms for SATISFIABILITY due to Dantsin, Goerdt, Hirsch and Schöning (2000) with a non-trivial upper bound on the time as a function of the number of variables n , for the case where the number of literals in every clause is bounded by a constant k . We shall in fact focus on $k = 3$, i.e., 3SAT. Note that the problem remains **NP**-complete with this restriction. As mentioned in Section 1, there are no known algorithms with a non-trivial upper bound as a function of n without such a restriction.

```

boolean SatBall(CNFformula  $f$ , non-negative integer  $d$ , TruthAssignment  $a$ )
  if  $a$  satisfies  $f$  then
    return true
  elseif  $d = 0$  then
    return false
  else
    Pick a clause  $C$  of  $f$  not satisfied by  $a$ ;
    Let  $x_1, \dots, x_k$  be the variables occurring (negated or unnegated) in  $C$ ;
    for  $i:=1$  to  $k$  do
       $a_i := a$  with the truth value assigned to  $x_i$  flipped
      if SatBall( $f, d - 1, a_i$ ) then
        return true
      fi
    od;
  return false
fi

```

Figure 5: The SatBall method.

The central idea in the algorithms we shall discuss is the use of the auxiliary method SatBall of Figure 5. The method takes as input a CNF formula f , a non-negative integer d and a full truth assignment to the variables of f . The method returns true if and only if f can be satisfied by some truth assignment *which can be obtained by flipping the truth values of at most d variables in a* . The method works by first checking if the answer is immediately **true** or **false** given a and d and otherwise finding a clause C in f which is not satisfied by a , and recursively calling itself on assignments obtained by flipping the truth value of a variable in f and with a decremented value of d .

The reason for the name SatBall is that we can define a *metric* (i.e., a distance function) on the set of truth assignments to variables in f by defining the distance between two truth assignments to be the minimum number of truth values that must be flipped to convert the first truth assignment into the other. Then SatBall(f, d, a) decides if there is a truth assignment to f in the *Ball* with center a and radius d , with respect to this metric.

The correctness of SatBall is given by the following lemma.

Lemma 8 SatBall(f, d, a) *correctly decides if f is satisfied by some truth assignment which can be obtained by flipping at most d truth values of a .*

Proof The proof is by induction in d . First, if $d = 0$, the statement is

easily seen to be correct. Now suppose $d > 0$ and assume that the statement is true for all smaller values of d .

First suppose that there is some truth assignment \tilde{a} which can be obtained by flipping at most d truth values of a that satisfies f . We shall argue that the algorithm returns **true**. If a itself satisfies f , the algorithm does return **true**. Otherwise, the algorithm picks some clause of f which is not satisfied by a , i.e., a makes all literals in the clause **false**. Since \tilde{a} does satisfy f it must make one of the literals in the clause, say x (which is either a negated or an unnegated variable) **true**. One of the recursive calls made by the algorithm is $\text{SatBall}(f, d - 1, a')$ where a' is as a but with the truth assignment to x flipped. As \tilde{a} can be obtained by flipping at most d truth values of a and one of these truth values is the truth value of x , \tilde{a} can be obtained by flipping at most $d - 1$ truth values of a' . Thus, $\text{SatBall}(f, d - 1, a')$ returns **true**, as desired.

Next suppose that there is no truth assignment which can be obtained by flipping at most d truth values of a satisfying f . We shall argue that the algorithm returns **false**. Suppose, to the contrary, that it returns **true**. It does so only if one of the recursive calls $\text{SatBall}(f, d - 1, a_i)$ returns **true**. But by induction, this is the case only if there is a satisfying assignment \tilde{a} to f within distance $d - 1$ from a_i . But a_i has distance 1 from a , and hence \tilde{a} has distance at most d from a , a contradiction. **QED**

If we restrict the input to clauses of size at most 3 (i.e., 3SAT instances), the worst case running time of $\text{SatBall}(f, d, a)$ is easily upper bounded by

$$T(d, l) \leq 3^d l^{O(1)},$$

where l is the length of the encoding of f (we here assume that the value d given to the algorithm is at most n , so that the length of the encoding d is smaller than the length of the encoding of f).

We next describe how to use the SatBall method to obtain a full blown algorithm for SATISFIABILITY. For notational convenience, we assume that the variables of the CNF formula f given as input are x_1, x_2, \dots, x_n so that we can describe truth assignments as vectors $a \in \{0, 1\}^n$, the semantics being that a assigns **true** to x_i if $a_i = 1$ and a assigns **false** to x_i if $a_i = 0$.

Clearly, any truth assignment can be obtained by flipping at most n truth values in the assignment assigning **false** to every variable. Thus, the procedure Sat1 in Figure 6 correctly decides SATISFIABILITY. By the complexity bound for SatBall given above, we have that the time complexity of Sat1 on 3SAT instances is bounded by $T(n, l) = 3^n l^{O(1)}$. Note that this is worse than the bound $2^n l^{O(1)}$ obtained by the naive SATISFIABILITY algorithm

```

boolean Sat1(CNFformula  $f$ )
  return SatBall( $f$ ,  $n$ , (0,0,...,0))

```

Figure 6: Using the SatBall method once.

```

boolean Sat2(CNFformula  $f$ )
  if SatBall( $f$ ,  $\lfloor n/2 \rfloor$ , (0,0,...,0)) then
    return true
  else
    return SatBall( $f$ ,  $\lfloor n/2 \rfloor$ , (1,1,...,1))
  fi

```

Figure 7: Using the SatBall method twice.

described in Section 1 and thus not competitive.

The key to improvement is to use the SatBall method more than once. Clearly, any truth assignment a on n variables can be obtained *either* by flipping at most $\lfloor n/2 \rfloor$ truth values in the truth assignment assigning **false** to all variables (in the case of a assigning **false** to at least half the variables) *or* by flipping at most $\lfloor n/2 \rfloor$ truth values in the truth assignment assigning **true** to all variables (in the case of a assigning **false** to less than half the variables). Thus, the algorithm Sat2 described in Figure 7 correctly decides SATISFIABILITY. Furthermore, the complexity of Sat2 on 3SAT instances is given by

$$T(n, l) \leq 2 \cdot 3^{\lfloor n/2 \rfloor} l^{O(1)} \leq 2 \cdot 3^{n/2} l^{O(1)} \leq 2^{0.79249n} l^{O(1)}$$

i.e. a significant improvement over the naive algorithm!

Based on the experience so far, it does not seem too surprising that we can obtain further improvements by using the SatBall method more than twice on carefully selected truth assignments and with a smaller value of d than $\lfloor n/2 \rfloor$. The question is how to select appropriate truth assignments. Suppose we end up using truth assignment $a_1, a_2, \dots, a_r \in \{0, 1\}^n$ and decide SATISFIABILITY for f by running SatBall(f, a_i, d) for $i = 1, \dots, k$ for some value of d and returning **true** if one of the calls to SatBall returns **true**. A necessary and sufficient condition for correctness of our algorithm is then given by

The Covering Property Any truth assignment $a \in \{0, 1\}^n$ is within distance d of one of a_1, a_2, \dots, a_r .

```

boolean CoveringCodeSat(CNFformula  $f$ )
  for  $a \in C$  do
    if SatBall( $f, d, a$ ) then
      return true
    fi
  od
  return false

```

Figure 8: The CoveringCodeSat method using code C of covering radius d .

Fortunately, families of vectors a_1, a_2, \dots, a_r satisfying the Covering Property are well studied in mathematics under the name of *covering codes*.

Definition (Covering Code) A covering code C over $\{0, 1\}^n$ with covering radius d is a set C of vectors in $\{0, 1\}^n$ so that for all vectors $a \in \{0, 1\}^n$, there is $b \in C$, so that a can be obtained by flipping at most d bits in b . The elements of C are called the *code words* of the code.

Given a covering code C over $\{0, 1\}^n$ with covering radius d , let CoveringCodeSat be the algorithm defined in Figure 8. The preceding discussion immediately implies that it correctly solves SATISFIABILITY. Furthermore, the complexity of running CoveringCodeSat on 3SAT instances with n variables is given by

$$T(n, l) \leq |C|3^d l^{O(1)}$$

We now simply have to consult the coding theory literature to find appropriate covering codes making this expression as small as possible. The following Theorem 9 is given without proof.

Theorem 9 *For any constants $0 < \epsilon, \rho < \frac{1}{2}$ and any sufficiently large integer $n \geq 1$, there is a covering code $C_{\epsilon, \rho, n}$ with at most $2^{(1-H(\rho)+\epsilon)n}$ code words and with covering radius at most $(\rho + \epsilon)n$. Furthermore, for any constants $0 < \epsilon, \rho < \frac{1}{2}$, there is a deterministic algorithm which on input n outputs a list of all the code words of $C_{\epsilon, \rho, n}$ and does so in time $2^{(1-H(\rho)+\epsilon)n} n^{O(1)}$.*

In Theorem 9, the function $H : [0, 1] \rightarrow [0, 1]$ is the *entropy* function $H(p) = -p \log_2 p - (1 - p) \log_2 (1 - p)$. As seen by easy calculus, H is monotonically increasing in the interval from 0 to $\frac{1}{2}$, it is monotonically decreasing in the interval from $\frac{1}{2}$ to 1 and has $H(0) = H(1) = 0$ and $H(\frac{1}{2}) = 1$. The code of Theorem 9 is almost optimal in the following sense: Any covering code over

$\{0, 1\}^n$ with covering radius ρn has at least $2^{(1-H(\rho))n}$ codewords (Exercise 3.1).

We now get the following theorem.

Theorem 10 *For any constant $\epsilon > 0$, there is an algorithm solving 3SAT instances with n variables and encoding length l in time $(\frac{3}{2} + \epsilon)^n l^{O(1)}$. In particular, there is an algorithm solving 3SAT in time $2^{0.58497n} l^{O(1)}$.*

Proof The algorithm is the CoveringCodeSat method of Figure 8 with covering code $C = C_{\epsilon, \frac{1}{4}, n}$.

Note that the number of code words of $C_{\epsilon, \frac{1}{4}, n}$ is

$$\begin{aligned} |C_{\epsilon, \frac{1}{4}, n}| &\leq 2^{(1-H(\frac{1}{4})+\epsilon)n} \\ &= 2^{(1+\frac{1}{4}\log_2(\frac{1}{4})+(1-\frac{1}{4})\log_2(1-\frac{1}{4})+\epsilon)n} \\ &= (2(\frac{1}{4})^{\frac{1}{4}}(\frac{3}{4})^{\frac{3}{4}}2^\epsilon)^n \end{aligned}$$

so by the previously given bound, the complexity on 3SAT instances is

$$\begin{aligned} T(n, l) &\leq (2(\frac{1}{4})^{\frac{1}{4}}(\frac{3}{4})^{\frac{3}{4}}2^\epsilon)^n 3^{(\frac{1}{4}+\epsilon)n} l^{O(1)} \\ &= (2(\frac{1}{4})^{\frac{1}{4}}(\frac{3}{4})^{\frac{3}{4}}3^{(\frac{1}{4})}6^\epsilon)^n l^{O(1)} \\ &= (2(\frac{3}{4})6^\epsilon)^n l^{O(1)} \\ &\leq (\frac{3}{2} + 3\epsilon)^n l^{O(1)} \end{aligned}$$

where the last inequality holds for sufficiently small ϵ . As $\epsilon > 0$ is arbitrary, we are done. **QED**

The best known worst case time bound for a deterministic algorithm for 3SAT is based on a refinement of the algorithm of Theorem 10 and achieves a running time of $2^{0.56658n} l^{O(1)}$. It obtains the improvement by making a somewhat more intelligent search in the neighborhood of each code word in the covering code than the search performed by the SatBall method.

Interestingly, there are *randomized* algorithms for 3SAT that are guaranteed to find a satisfying assignment to any satisfiable 3SAT instance given as input in *expected* time roughly $2^{0.4n}$, i.e., significantly better than $2^{0.56658n} l^{O(1)}$.

3.1 Exercises

Exercise 3.1 Let $\rho \leq \frac{1}{3}$ and $n \geq 4$. Show that any covering code over $\{0, 1\}^n$ with covering radius ρn has at least $2^{(1-H(\rho))n}$ code words. The following useful inequality may be used without proof: For all positive integers m, n with $2 \leq m \leq \frac{n}{2}$, it holds that $\binom{n}{m} \leq \frac{1}{2} 2^{H(m/n)n}$.

Exercise 3.2 Why is ρ set to $\frac{1}{4}$ in the proof of Theorem 10? Is this the best possible value for the application?

Exercise 3.3 Show that, for any $\epsilon > 0$, there is an algorithm for k SAT running in time $(\frac{2k}{k+1} + \epsilon)^n l^{O(1)}$.

4 Practical issues

The worst case bounds given in the previous sections indicate that it is possible to handle instances of SATISFIABILITY whose size are bigger by a small constant factor than those one can handle by a naive algorithm. However, for the desired real-life applications of SATISFIABILITY, such as hardware verification where the number of variables may be counted in thousands, this is not enough.

What one can hope for and what indeed turns out to be true in many cases is that the instances of those applications possess some structure that make them easier than worst case instances, so that they can be handled by variations of the algorithms we have seen, or other algorithms, in reasonable time.

Thus, we are hoping for a situation somewhat similar to the situation of the *linear programming* problem and the *simplex algorithm*, namely, that a worst case exponential algorithm will actually turn out to be adequate in practice. But one should realise that this analogy is somewhat misleading: In the case of the Simplex algorithm, the instances with the exponential behavior are rare and artificial. Furthermore, one needs different examples for different versions of the simplex algorithm. Thus, for linear programming, there are no “universally” hard instances and indeed, it is widely conjectured that a randomized version of the simplex algorithm will actually handle all instances in expected polynomial time. And, if not, we do have the interior point methods to go back to. On the other hand, in the case of Satisfiability, its **NP**-completeness seems to cause difficult instances to be the rule rather than the exception. Thus, one easily can come up with instances which no known algorithm are able to handle efficiently, such as instances corresponding to factoring big numbers, serving as public RSA keys. Furthermore, we expect (and the cryptography industry hopes) that *no* algorithm, known or not known, can handle these instances efficiently. Therefore, we expect the practicality of our SATISFIABILITY algorithms to be much more restricted than the practicality of the simplex algorithm. Also, we can expect that different families of “easy” instances may need different approaches.

Lacking theoretical bounds, obtaining sound experimental data on the performance of various approaches becomes essential. Doing so in a sound way is methodologically non-trivial. There are (at least) three approaches that have been followed in the literature. While the SATISFIABILITY case is a very well studied one and perhaps the case for which most studies are available, these approaches are of course not limited to SATISFIABILITY but

can be used to experimentally study algorithms for any hard problem.

- *Case story approach.* The basis for this approach is a single SATISFIABILITY instance or a family of such instances that has arisen in some particular context. At the beginning we do not know whether the instance is satisfiable or not. Furthermore, the instances are too big to solve by a naive approach. Real examples are instances whose satisfiability would mean that a particular piece of hardware is *not* correct and instances for which a satisfying assignment corresponds to proofs of mathematical theorems that have not yet been established. By tailoring our SATISFIABILITY algorithm to do well on the instance, we manage to solve it and report how we did so.
- *Benchmark approach.* For evaluating algorithms which are intended to be more generally applicable, there are libraries available containing solved SATISFIABILITY instances from various sources and various levels of difficulty. On the web, links can be found, for instance, at <http://www.satlib.org> and <http://www.cs.toronto.edu/~kullmann/SATlinks.html>
- *Random instance approach.* The case story and benchmark approaches do not really allow for doing experiments with a very meaningful statistical conclusion. Thus, often experimenters turn to randomly generated instances instead, where the instances are generated according to a probability distribution that is (hopefully) tuned so that the generated instances somehow resemble the real instances we hope to solve. With such a well defined source, it is possible to obtain reliable statistical information, but of course only on how well the algorithm behaves on instances from the source. In fact, sometimes we may be able to obtain the information *without* doing experiments by a theoretical average case analysis. But it is not obvious how to directly relate the information obtained to real life performance.

In this section, we outline two practical approaches to SATISFIABILITY solving that have been relatively successful in the various kinds of experiments considered above. One is the Davis-Putnam procedure which we studied above from a theoretical perspective and the other is the very different Canonical Form approach. This enumeration of practical approaches to SATISFIABILITY solving is by no means exhaustive. One approach which we shall cover only implicitly is the transformation of the SATISFIABILITY instance to an instance of another **NP**-hard problem which we then solve

using software for that problem. Theoretically, any **NP**-hard problem can of course be used, by the definition of **NP**-hardness. A problem for which such a transformation becomes of practical importance and becomes a truly different way of solving SATISFIABILITY instances, is INTEGER LINEAR PROGRAMMING. We shall study algorithms for INTEGER LINEAR PROGRAMMING later in the course. Common to all the approaches is that they are only heuristics as far as the *running time* of the algorithm is concerned: They all correctly decide satisfiability. Later in the course, when we discuss local search heuristics, we shall see examples of heuristics for SATISFIABILITY and other search and optimization problems where also the correctness of the output is not guaranteed. Clearly, the methodological problems there does not become smaller.

For more information and links to literature, experimental studies and software for SATISFIABILITY solving, we recommend the web page of Oliver Kullman:

<http://www.cs.toronto.edu/~kullmann/SATlinks.html>

4.1 Davis-Putnam variations

We presented various versions of the Davis-Putnam procedure with non-trivial worst case running bounds in Section 2. The procedure is also very important from a practical perspective. For practical Davis-Putnam implementation, there are some issues that were ignored in the worst case analysis above that becomes relevant.

The polynomial overhead

In the theoretical analysis, we expressed the complexity as $2^{\alpha m}l^{O(1)}$ and concentrated on optimizing α with no concern about the polynomial overhead. In practical algorithms, making the overhead as small as possible is a prime issue. Thus, the design and analysis of appropriate data structures for representing CNF formula and truth assignments so that the simplification routine can be performed efficiently has been a major theme. For practical implementations, the experience is that it is worth sacrificing rewrite rules that are rarely used if this reduces the overhead, even if the asymptotic worst case performance becomes worse. We shall not dwell on these issues here. A work where the importance of appropriate data structures is carefully documented is the SATO project by Zhang that uses *tries* for representing CNF formulas (<http://www.cs.uiowa.edu/~hzhang/sato.html>).

Selecting the variable on which to branch

Another very important issue in the Davis-Putnam method is the selection of the variable on which to branch. In the theoretical worst case analysis we saw that the appropriate strategy was to select the variable that reduce the number of clauses in $\text{Simplify}(f[x \leftarrow \mathbf{false}])$ and $\text{Simplify}(f[x \leftarrow \mathbf{true}])$ as much as possible. This is also true from a practical perspective. Indeed, the reason that the procedure is often practical, even on large instances, is that it is often possible to remove many more clauses than the worst case analysis suggests. This also explains why the more esoteric rewrite rules that helped the worst case analysis are of little practical importance - the bottleneck situations that they are supposed to get rid of simply do not occur that often. However, the worst case analysis does suggest a good heuristic rule for selecting which variable to branch (suggested by Kullman):

Kullman Selection Rule For each variable x , let p_x be the number of clauses removed in $\text{Simplify}(f[x \leftarrow \mathbf{false}])$ and q_x be the number of clauses removed in $\text{Simplify}(f[x \leftarrow \mathbf{true}])$. Furthermore, let v_x be the positive real root of $t \rightarrow 1 - 1/t^{p_x} - 1/t^{q_x}$. Select the variable x minimizing v_x .

The motivation behind the rule is that *if* we succeed in picking a variable with $v_x \leq \alpha$ each time we branch, the running time of our algorithm will at most be proportional to α^m . Intuitively, if we succeed in finding variables x with $v_x \leq \alpha$ in most but not all cases, we won't do much worse. Thus, there is good theoretical motivation for using Kullman's selection rule. On the other hand, the overhead for using the rule is big, as we have to "search one move ahead" and consider the result of branching on each of the n different variables.

This can be remedied as follows: Let q'_x be the number of occurrences of x in the formula under consideration and let p'_x be the number of occurrences of $\neg x$. Since we know that we will remove *at least* p'_x clauses in $\text{Simplify}(f[x \leftarrow \mathbf{false}])$ and *at least* q'_x clauses in $\text{Simplify}(f[x \leftarrow \mathbf{true}])$, the following rule makes some sense:

Modified Kullman Selection Rule For each variable x , let q'_x be the number of occurrences of x in the formula under consideration and let p'_x be the number of occurrences of $\neg x$. Furthermore, let v_x be the positive real root of $t \rightarrow 1 - 1/t^{p'_x} - 1/t^{q'_x}$. Select the variable x minimizing v_x .

The Modified Kullman Selection Rule has much smaller overhead. Note that one would not compute the real root of $t \rightarrow 1 - 1/t^{p'_x} - 1/t^{q'_x}$ numerically on the fly, but rather, build into the program a table similar to (but bigger than) the table of Figure 3.

The following rule, due to Freeman, is somewhat simpler but behaves in a similar way (Exercise 4.1).

Freeman Selection Rule For each variable x , let q'_x be the number of occurrences of x in the formula under consideration and let p'_x be the number of occurrences of $\neg x$. Furthermore, let $w_x = (1 + q'_x)(1 + p'_x)$. Select the variable x maximizing w_x .

Fine-tuning the selection rule for particular instances is an important part of the “case study” approach to experimental studies of the Davis-Putnam procedure.

“Intelligent backjumping”

The Davis-Putnam procedure on input formula f performs two recursive calls, one on $f[x \leftarrow \mathbf{false}]$ and one on $(f[x \leftarrow \mathbf{true}])$. It would improve total efficiency if information gained from one recursive call could improve the efficiency of the search in the other recursive call.

“Intelligent backjumping” is a technique that sometimes makes it possible to eliminate the second recursive call completely. Suppose $\text{DavisPutnam}(f[x \leftarrow \mathbf{true}])$ returns **false**. During the execution of $\text{DavisPutnam}(f[x \leftarrow \mathbf{true}])$, other variables of f than x were perhaps also assigned values. This happened either because we branched on those variables or because rewrite rules, such as Rule IV, caused the value of the variable to be fixed. Now suppose we, in some data structure keep track of which variables *influenced* which clauses and variables. For instance, if f contains the clause $(\neg x \vee y)$ and we call $\text{DavisPutnam}(f[x \leftarrow \mathbf{true}])$, Rule III will make the clause $(\neg x \vee y)$ into (y) so we will say that x influenced that clause. Then Rule IV will make y **true** and we will say that x influenced y . If y then influences some other variables and clauses, we will also say that x influences those. Thus, x may influence many variables and clauses. On the other hand, making x **true** could also have very little effect on the formula, influencing only a few variables and clauses. Doing such bookkeeping carefully, it is possible to call $\text{DavisPutnam}(f[x \leftarrow \mathbf{true}])$, get the result **false** and also get some information back that may indicate that the variable x *did not influence the result*, i.e., that the result would have been false, even if x had been **false**. Then, we do not have to do the second recursive call. Whether or not the bookkeeping is worth doing depends on how big the overhead is and how often we would expect the situation described above to occur in practice. In some situations, considerable efficiency gains have been reported.

The idea of making the search in one recursive branch influence the search

in another can be carried out in a much more systematic and clean way for many optimization problems using a technique called *branch and bound* which will be covered later in the course.

Constraint satisfaction languages

The implementation of *Constraint satisfaction languages* can be viewed as variations of the Davis-Putnam procedure but with a somewhat wider perspective.

A constraint satisfaction language is a language for expressing sets of constraints for some object to be found. An example of a constraint satisfaction language is thus SATISFIABILITY with the constraints being the clauses and the object to be found a truth assignment. By Cook's theorem, any constraint satisfaction problem with constraints that can be easily checked for a given object can be encoded as a SATISFIABILITY instance. But this may not be very convenient encoding for a particular search problem instance. In particular, if we allow variables which are not just Boolean but with a bigger range and also allow the use of constraints which are not just disjunctions, perhaps we would be able to express our problem instance more concisely and naturally, using fewer variables and constraints. Thus, it makes sense to consider a richer language than SATISFIABILITY. To solve the instances represented in the enriched language, we should then make a modification of the Davis-Putnam procedure adapted for the richer language. A modification we would have clearly have to make is that we cannot just make two recursive calls when we branch on a non-Boolean variable. The necessary modification is not necessarily obvious: For a many-valued variables, we may have to do something more intelligent than just make a recursive call on each possible value. The most important issue turns out to be how to make an appropriate Simplify procedure. In the terminology of constraint satisfaction language, the analogy of the Simplify procedure is called *constraint propagators*.

Ideally, one wants a language where the problem instance under consideration can be expressed naturally and concisely and a language which possesses a set of constraint propagators which makes the problem instance efficiently solvable. These two objectives of course interfere and one should not for general applications, expect "optimal" languages. A lot of research on constraint satisfaction languages consist in finding good languages for particular, specialized problem domains and sets of problems.

4.2 Canonical form algorithms and OBDD's

One way of looking at the SATISFIABILITY problem is as follows. We are given a Boolean formula over some set of variables. Assume without loss of generality that the variables are named x_1, x_2, \dots, x_n . Then the formula describes a Boolean *function* $f : \{0, 1\}^n \rightarrow \{0, 1\}$ in the obvious way, where we, for convenience, represent **false** as 0 and **true** as 1. The problem is to decide if this function is identically zero.

This way of looking at the problem leads to the canonical form approach to solving it. Suppose we have some way of representing Boolean functions as strings over an alphabet Σ , i.e. a coding scheme e , so that

1. Each Boolean function has only one encoding, i.e., e is a map, mapping the set of Boolean functions to Σ^* .
2. Given the encoding of f and g , we can efficiently compute the encoding of $\neg f$, $f \vee g$ and $f \wedge g$. More precisely, there are polynomial procedures NOT, AND and OR, so that $\text{OR}(e(f), e(g))=e(f \vee g)$, $\text{AND}(e(f), e(g))=e(f \wedge g)$ and $\text{NOT}(e(f))=e(\neg f)$.

We call such a coding scheme e a *canonical coding scheme* and we call $e(f)$ the *canonical form* of f . Note that one obvious way of representing Boolean functions, namely, representing them as (encodings of) Boolean formulas is *not* a canonical coding scheme, as many Boolean formulas may represent the same Boolean function. Another obvious way of representing Boolean functions, namely, representing them as encodings of their *truth tables* (for instance as a 2^n bit Boolean string), *is* an canonical coding scheme as each Boolean function only has one truth table representing it. However, this canonical encoding is useless for our purposes, as every Boolean function on n variables is represented by an exponentially long string. What we need is a coding scheme where we can hope that the functions we actually encounter have a short code.

We will come back in a minute to an coding scheme that is canonical *and* useful but let us first see the relevance of the whole setup by deriving a SATISFIABILITY algorithm, given a canonical coding scheme. In Figure 9 such an algorithm is given. The procedure CanonicalFormSat decides satisfiability of a Boolean formula, not necessarily in conjunctive normal form, by computing the canonical encoding of the function corresponding to the formula using the auxiliary method CanonicalForm and simply comparing it to the encoding of the all-zero function, denoted by $v \rightarrow 0$. The CanonicalForm method computes the canonical form of the function corresponding to

```

String CanonicalForm(Boolean formula  $f$ )
  if  $f$  is a single variable  $x_i$  then
    return  $e(v \rightarrow v_i)$ 
  elseif  $f = \neg g$  for some  $g$  then
    return NOT(CanonicalForm( $g$ ))
  elseif  $f = g \vee h$  for some  $g, h$  then
    return OR(CanonicalForm( $g$ ), CanonicalForm( $h$ ))
  elseif  $f = g \wedge h$  for some  $g, h$  then
    return AND(CanonicalForm( $g$ ), CanonicalForm( $h$ ))
  fi

boolean CanonicalFormSat(Boolean formula  $f$ )
  return CanonicalForm( $f$ )  $\neq e(v \rightarrow 0)$ 

```

Figure 9: The CanonicalFormSat method with canonical coding scheme e .

the formula using a simple recursive method utilising the procedures NOT, AND, OR which are guaranteed to exist by the definition of a canonical coding scheme.

The complexity of the CanonicalFormSat procedure obviously depends on the canonical coding scheme used. There is no known coding scheme that makes the procedure polynomial (this would imply $\mathbf{P} = \mathbf{NP}$). A good coding scheme should make the procedure exponential in the worst case, but, hopefully, faster on the instances of interest.

An approach that does lead to a canonical coding scheme is the following: First, we can represent a function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ by just representing the subset $f^{-1}(1) \subseteq \{0, 1\}^n$. Next, note that since $f^{-1}(1)$ is *finite*, it is also a *regular* set, i.e., we can represent it by a *finite automaton* deciding it. There may be many such automata, but suppose we pick one with the minimum number of states. Then, a famous result of automata theory (the Myhill-Nerode theorem) tells us that this minimum-state automaton is unique, up to isomorphism (i.e. renaming of states). Thus, if we can find an encoding of finite automata so that isomorphic automata get the same encoding (and such an encoding is relatively straightforward to obtain, see exercise 4.2), we can define the encoding of f to be the encoding of a minimum-state finite automaton deciding $f^{-1}(1)$. The uniqueness of the minimum-state automaton ensures that we have the desired property 1 of a canonical encoding. Also, there is a polynomial time procedure, which, given a finite automaton outputs an equivalent minimum-state automaton in polynomial time. This, together

with simple automata theoretic constructions ensures that our encoding also satisfies property 2.

The canonical automata encoding is very close to the encodings that are actually used in practice, for such applications as hardware verification. In particular, a very popular canonical coding scheme for Boolean functions is the *ordered binary decision diagram* or *OBDD*. An OBDD is, essentially, a finite automata without cycles in the underlying graph. This restriction is convenient as the languages we look at are finite and all strings in the language has the same length. The restriction makes the AND and OR procedures much more efficient and practical.

One advantage of the canonical form approach is that it can be *directly* used not only to solve SATISFIABILITY but also to solve a wide range of other problems concerning Boolean functions and their representations which are very useful for verification applications. Most of these problems being in **NP** or **coNP** we could of course use any algorithm for SATISFIABILITY to *indirectly* solve them, but such an approach would be less practical.

One disadvantage of the canonical form approach is that it not only has exponential *running time* in the worst case, it also uses an exponential amount of *memory space*. In fact, the time usage is polynomial in the space usage. Compare this with the Davis-Putnam procedure whose space usage is polynomial in the size of the original instance (Exercise 4.6). This has the practical implication that while the Davis-Putnam procedure can usually be carried out with all data in the fast *cache* memory of the computer, even for worst case instances, canonical form algorithms may need disk swapping. This may deteriorate their performance by orders of magnitude.

4.3 Exercises

Exercise 4.1 Use Table 3 to compare the Modified Kullman Selection Rule and the Freeman Selection Rule for p'_x, q'_x between 1 and 10. Do they typically make the same decision? Are there examples where they make very different decisions?

Exercise 4.2 Show that there is an efficient encoding scheme e , mapping finite automata to strings over a finite alphabet, so that isomorphic automata get the same encoding and non-isomorphic automata do not.

Exercise 4.3 Show that there is a canonical coding scheme making CanonicalFormSat into a polynomial procedure if and only if **P=NP**.

Exercise 4.4 What does the encoding of the all-zero function $v \rightarrow 0$ look like in the finite automaton canonical coding scheme? What about the encoding of the function $v \rightarrow v_i$?

Exercise 4.5 Show that for the finite automaton canonical coding scheme, the method AND can be implemented so that it runs in polynomial time. You may use without proof that there is a polynomial procedure minimizing finite automata.

Exercise 4.6 Show that the time use of a canonical form algorithm solving SATISFIABILITY is polynomial in the space use. Show that the space use of the Davis-Putnam procedure is polynomial in the size of the original instance.

Exercise 4.7 Would the covering code based satisfiability algorithm of Section 3 be an appropriate basis for a practical solver? Discuss.