Guided Search and a Faster Deterministic Algorithm for 3-SAT

Dominik Scheder*

Theoretical Computer Science, ETH Zürich
CH-8092 Zürich, Switzerland
dscheder@inf.ethz.ch

Abstract. Most deterministic algorithms for NP-hard problems are *splitting algorithms*: They split a problem instance into several smaller ones, which they solve recursively. Often, the algorithm has a choice between several splittings. For 3-SAT, we show that choosing wisely which splitting to apply, one can avoid encountering too many worst-case instances. This improves the currently best known deterministic worst case running time for 3-SAT from $\mathcal{O}(1.473^n)$ to $\mathcal{O}(1.465^n)$, n being the number of variables in the input formula.

1 Introduction

Most deterministic algorithms for NP-hard problems like k-SAT, k-colorability and Maximum Independent Set use the idea of splitting: A problem instance I is replaced by several smaller instances $I_1, \ldots, I_\ell$, which are solved recursively. Of course, we want ℓ to be small and the size of the instances I_i to be much smaller than the size of I—whatever *size* means in this context. Most algorithms use several branching rules, i.e. rules for replacing I by $I_1, \ldots, I_\ell$. Inevitably, not every rule will apply to every instance, and some rules will amount to higher running time and some to lower. Often, a single rule is responsible for the worst-case behavior of the algorithm. Imagine you have a "meta-rule" that tells you what branching rule to apply in order to avoid encountering too many worst-case instances. This will of course speed up your algorithm. For our 3-SAT-algorithm, we find such a meta-rule. The general idea is to run a preliminary search on a given instance I that simply aborts when a worst case instance is encountered. We pick *one* such worst-case instance I' and again start the preliminary search on I'. Repeating, we will find an instance I^* and a search tree for I^* that contains no worst-case instances. The trick is that one can show, for our particular algorithm, that this very search tree is also a search tree for I. We use the instance I^* as a *search guide* for I, always applying the branching rules that would have applied in the search tree for I^*. The algorithm to which we apply this idea is the deterministic local search algorithm for 3-SAT by Dantsin et al. [3], of which we improve the running time from the previously best known $\mathcal{O}(1.473^n)$ [2] to $\mathcal{O}(1.465^n)$ (here, n is the number of variables). The idea is in fact not limited to 3-SAT, it can be applied for general k-SAT, but the improvement over the original algorithm by [3] becomes smaller and smaller.

* Research is supported by the SNF Grant 200021-118001/1.

E.S. Laber et al. (Eds.): LATIN 2008, LNCS 4957, pp. 60–71, 2008.

Previous Results

In recent years, a lot of research has been done in designing "modestly exponential" algorithms deciding 3-SAT, i.e. running in time $\mathcal{O}(a^n)$, for a considerably smaller than 2. The currently fastest *randomized* algorithm, given by Daniel Rolf [7], achieves a running time of $\mathcal{O}(1.32216^n)$.

The running times of *deterministic* algorithms for 3-SAT are much higher: Dantsin et al. [3] gave a deterministic algorithm based on local search, with a running time of $\mathcal{O}(1.481^n)$. Later, Brueggemann and Kern [2] further improved this algorithm and obtained a running time of $\mathcal{O}(1.473^n)$, which was the previously fastest known deterministic algorithm.

We apply the idea of *guided search* to the splitting algorithm in Dantsin et al. [3] and Brueggemann and Kern [2], thus avoiding encountering too many worst-case formulas and improving the running time of deterministic local search algorithms for 3-SAT from $\mathcal{O}(1.473^n)$ to $\mathcal{O}(1.465^n)$.

Notation

A CNF formula, or simply a CNF, is a conjunction (`and`) of clauses, and a clause is a disjunction (`or`) of literals. A literal is either a boolean variable x or its negation $\bar{x}$. We can assume that no clause contains both a variable and its negation. A k-CNF is a CNF in which every clause contains at most k literals, and k-SAT is the problem whether a given k-CNF is satisfiable. If γ is a *partial* truth assignment, then we denote by $F^{[\gamma]}$ the k-CNF obtained by setting the variables of F as described by γ. If γ does not set variable x, we may write $[\gamma, x \mapsto 1]$ (or $[\gamma, x \mapsto 0]$) to denote the partial assignment that behaves like γ, and in addition sets x to 1 (or to 0).

2 The Local Search Algorithm *k*-SAT

In [3], Dantsin et al. give a surprising approach to deciding k-SAT. Let F be a k-CNF and n be the number of variables in F. Let $\{0,1\}^n$ be the set of all possible truth assignments to these variables. We search for a satisfying assignment not in the whole cube $\{0,1\}^n$, but locally in some *Hamming ball* $B_r(\alpha) := \{\beta \in \{0,1\}^n : \ d(\alpha,\beta) \leq r\}$ of radius r centered at some $\alpha \in \{0,1\}^n$. We say F is $B_r(\alpha)$-*satisfiable* if $B_r(\alpha)$ contains an assignment satisfying F. We will see below how this can be decided for k-CNFs in time $\mathcal{O}(k^r \mathrm{poly}(n))$. For certain values of r, k^r is much smaller than the volume of $B_r(\alpha)$. By choosing $N(n,r)$ many Hamming balls that together cover $\{0,1\}^n$, we can decide satisfiability of F in time $\mathcal{O}(N(n,r)k^r \mathrm{poly}(n))$. There is of course a trade-off between the radius r of the balls and the number of balls needed to cover $\{0,1\}^n$. Dantsin et al. [3] show how to choose r optimally such that if $B_r(\alpha)$-satisfiability can be decided in $\mathcal{O}(a^r \mathrm{poly}(n))$, satisfiability of F can be decided in $\mathcal{O}\left(\left(2 - \frac{2}{1+a}\right)^n \mathrm{poly}(n)\right)$.

Note that by the symmetry of $\{0,1\}^n$, $B_r(\alpha)$-satisfiability is basically the same problem for each $\alpha \in \{0,1\}^n$. Hence we will assume for the rest of the paper that $\alpha = (1,\ldots,1)$ and write B_r for $B_r\big((1,\ldots,1)\big)$. Algorithm 1, given in Dantsin et al. [3], decides B_r-satisfiability in $\mathcal{O}(k^r \mathrm{poly}(n))$ steps.

Algorithm 1. `searchball`(Formula F, depth r)

1: **if** F contains no negative clause **then**
2: **return** `true`
3: **else if** $\square \in F$ or $r \leq 0$ **then**
4: **return** `false`
5: **else**
6: pick some negative clause $\{\bar{x}_1,\ldots,\bar{x}_\ell\} \in F$
7: **return** $\bigvee_{i=1}^{\ell}$ `searchball`$(F^{[x_i \mapsto 0]}, r-1)$
8: **end if**

Here, a negative clause is a clause containing only negative literals (and thus is not satisfied by $\alpha = (1,\ldots,1)$). Let us see why this algorithm works. The first four lines should be clear: If F contains no negative clause, α satisfies F, and surely $\alpha \in B_r$. Otherwise, if $\square \in F$, then F is clearly unsatisfiable. Also, if α does not satisfy F, and $r = 0$, then F is clearly not B_r-satisfiable.

So much for the base cases. The interesting step is of course the recursion. Consider the negative clause $\{\bar{x}_1,\ldots,\bar{x}_\ell\}$. If there is some satisfying assignment $\alpha^* \in B_r$, it must set some x_i to 0. Let α_i^* be the assignment setting x_i to 1, but else agreeing with α^*. Since $d(\alpha^*,\alpha) \leq r$, it holds that $d(\alpha_i^*,\alpha) \leq r-1$. Note that α_i^* satisfies $F^{[x_i \mapsto 0]}$. Therefore, F' is B_{r-1}-satisfiable, and the recursive call `searchball`$(F^{[x_i \mapsto 0]}, r-1)$ will return `true`. Conversely, if some $F^{[x_i \mapsto 0]}$ is B_{r-1}-satisfiable, it is easy to see that F is B_r-satisfiable.

3 Branching Rules

We say in lines 6 and 7 `searchball` performs a branching. To be more precise, define branchings as follows:

Definition 3.1. *For a partial assignment γ, let $|\gamma|$ denote the number of variables γ sets to 0. A* branching for F *is a set $\Gamma = \{\gamma_1,\ldots,\gamma_\ell\}$ of partial assignments with $|\gamma_i| \geq 1$ for all $1 \leq i \leq \ell$.*

Note that we only count variables γ sets to 0. This has two reasons. First, almost no assignment we encounter sets any variable to 1. We will see an exception at the end of the paper, but this will not cause any trouble. Second, we want to measure how far a partial assignment takes us from $\alpha = (1,\ldots,1)$, and setting variables to 1 obviously does not increase the distance from α. The intuition behind the definition of branchings is that `searchball` first computes some branching Γ for F and then recurses on each of the formulas $F^{[\gamma]}$ for each $\gamma \in \Gamma$. The following definition and observation ensure that this is legal, i.e. will give a correct result.

Definition 3.2. *We define* valid *branchings inductively. Let F be a CNF. For every negative clause $\{\bar{x}_1, \ldots, \bar{x}_\ell\} \in F$ the branching*

$$\{[x_1 \mapsto 0], \ldots, [x_\ell \mapsto 0]\}$$

is valid for F. If some branching Γ is valid for F, and there is some $\gamma \in \Gamma$ and a branching $\Gamma' = \{\beta_1, \ldots, \beta_\ell\}$ that is valid for $F^{[\gamma]}$, then

$$\Gamma \setminus \gamma \cup \{\gamma\beta_1, \ldots, \gamma\beta_\ell\}$$

is valid for F.

Here, $\gamma\beta_i$ denotes the combination of both partial assignments. Note that this is well defined, as these two partial assignments refer to disjoint sets of variables. The following observation gives meaning to the previous definition.

Observation. If Γ is a branching valid for F, then F is B_r-satisfiable if and only if there exists some $\gamma \in \Gamma$ such that $F^{[\gamma]}$ is $B_{r-|\gamma|}$-satisfiable.

One might think that these definitions are overly formal, since the notion of branchings in the context of recursive algorithms is a familiar one. However, as our algorithm becomes more involved, it will become clear that it pays to have things defined formally. Using the definition of branchings, we can replace lines 6 and 7 by

6: apply some rule to obtain a valid branching $\{\gamma_1, \ldots, \gamma_\ell\}$ for F
7: **return** $\bigvee_{i=1}^{\ell}$ `searchball`$(F^{[\gamma_i]}, r - |\gamma_i|)$

It is clear that for 3-CNFs, we always find a valid branching containing at most 3 partial assignments, thus `searchball` has a running time of $\mathcal{O}(3^r \mathrm{poly}(n))$. Our goal is to decrease the basis of the exponential to some $a < 3$. To achieve this, we first give four relatively simple branching rules for 3-CNFs.

3.1 Simple Branching Rules

Let $\mathrm{Neg}(F)$ denote the set of all negative clauses in F, i.e. clauses without positive literals. Accordingly, the empty clause $\square$ is a negative clause, too. If $\mathrm{Neg}(F)$ consists of pairwise disjoint clauses, we say F is Neg-*disjoint*. From now on, we assume that all negative 3-clauses in F are pairwise disjoint, i.e. that F is Neg-disjoint or contains some negative clause of size at most two. We will show at the end of the paper how to deal with intersecting negative 3-clauses. Let us state four simple rules the algorithm tries to apply. See Figure 1 for an illustration.

Rule 1. If there is some $\{\bar{x}_1, \ldots, \bar{x}_\ell\} \in \mathrm{Neg}(F)$ with $\ell \leq 2$, then use the branching $\{[x_1 \mapsto 0], \ldots, [x_\ell \mapsto 0]\}$. This includes the case $\square \in F$.

Note that if Rule 1 does not apply, then by assumption F is Neg-disjoint. Clearly, any satisfying assignment needs to set at least $|\mathrm{Neg}(F)|$ variables to 0. Hence if $|\mathrm{Neg}(F)| > r$ at this point, the algorithm immediately returns "not B_r-satisfiable". We assume from now on that $|\mathrm{Neg}(F)| \leq r$.

Rule 2. Suppose F contains two clauses of the form $\{u\}$ and $\{\bar{u}, \bar{v}, \bar{w}\}$. Use the branching $\{[v \mapsto 0], [w \mapsto 0]\}$. Note that the partial assignment $[u \mapsto 0]$ need not be part of the branching, because $F^{[u \mapsto 0]}$ contains the empty clause.

Rule 3. Suppose F contains clauses of the form $\{u, \bar{a}\}$ and $\{\bar{u}, \bar{v}, \bar{w}\}$. Use the branching $\{[u \mapsto 0, a \mapsto 0], [v \mapsto 0], [w \mapsto 0]\}$.

Rule 4. Suppose F contains clauses of the form $\{u, x\}$, $\{\bar{x}, \bar{y}, \bar{z}\}$ and $\{\bar{u}, \bar{v}, \bar{w}\}$, the latter two being distinct. Use the branching $\{[u \mapsto 0, y \mapsto 0], [u \mapsto 0, z \mapsto 0], [v \mapsto 0], [w \mapsto 0]\}$. Similarly to the case for Rule 2, the partial assignment $[u \mapsto 0, x \mapsto 0]$ is not part of the branching, since $\square \in F^{[u \mapsto 0, x \mapsto 0]}$.

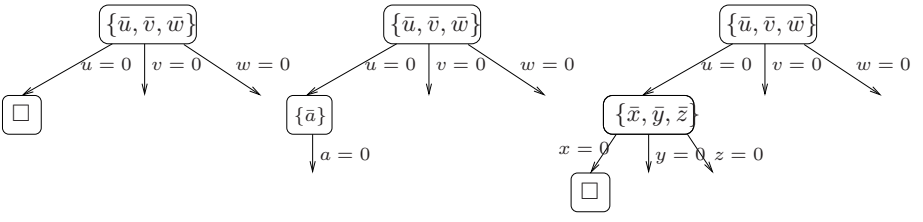

Fig. 1. Visualization of Rules 2, 3 and 4, respectively

It should be noted that there is nothing new about these branching rules. They all appear in [2], and some appear already in [3]. Each formula occurring in the computation of `searchball`(F,r) is of the form $F^{[\gamma]}$ for some partial assignment γ. In this sense, branching rules extend γ: For example, Rule 3 extends it to $[\gamma, u \mapsto 0, a \mapsto 0]$, $[\gamma, v \mapsto 0]$ and $[\gamma, w \mapsto 0]$ in the recursive calls.

4 Partial Exact Assignments and Guided Search

If some of Rules 1–4 applies to F, then `searchball` applies a branching Γ and calls itself recursively on $F^{[\gamma]}$ for each $\gamma \in \Gamma$. If none of these rules applies to F, we call F *reduced*. This is where the difficult part begins. The approach of [3] and [2] is (briefly) to define additional rules and then prove a non-trivial theorem that if these rules do not apply, there is some other way to decide quickly whether F is B_r-satisfiable. Unfortunately, the additional rule causes a higher running time. Our approach is not completely different, however, we do not introduce any additional rules. Observe that Rules 1–4 might give several

valid branchings for the same formula. It turns out that, depending on which branching we decide to apply, we may encounter reduced formulas very often or very rarely. We show that we can find a "guide" formula that tells us which branchings to apply in order to avoid encountering too many reduced formulas. Central to our algorithm will be the following special type of partial assignments:

Definition 4.1. *Let F be a* Neg*-disjoint CNF. A partial assignment γ to the variables of F is called a* partial exact assignment *w.r.t. F, short* pea*, if*

- *it sets no variable to* 1*, and*
- *in each clause $C \in \mathrm{Neg}(F)$, it satisfies at most one literal (i.e. it sets the corresponding variable to* 0*), and*
- *it does not set further variables.*

For example, if $F = \{\{\bar{u}, \bar{v}\}, \{\bar{x}, \bar{y}\}, \{x, \bar{a}\}\}$, then $\gamma = [u \mapsto 0, y \mapsto 0]$ is a pea for F, but $[u \mapsto 0, v \mapsto 0]$ and $[u \mapsto 0, a \mapsto 0]$ are not. Please note that though defined in more general terms, we will use the term pea w.r.t. F only if F is a reduced 3-CNFs. A crucial fact in our algorithm will be that Rules 2 and 4 behave "nicely" with respect to peas. This means, if γ is a pea w.r.t. some F, and Rule 2 or Rule 4 applies to $F^{[\gamma]}$, then the extensions of γ produced by the branching will be peas w.r.t. F, as well. This is because all variables set to 0 in Rule 2 and 4 occur in a 3-clause of $F^{[\gamma]}$. Since applying γ does not create new 3-clauses, these must already have been in F. However, Rules 1 and 3 can produce non-peas, as they set variables to 0 which do not necessarily occur in a 3-clause.

If we encounter a reduced formula F, we cannot apply any of Rules 1–4 and thus have to pick some $\{\bar{x}_1, \bar{x}_2, \bar{x}_3\} \in \mathrm{Neg}(F)$ and recurse on the three formulas $F^{[x_i \mapsto 0]}, i = 1, 2, 3$. This branching rule, if applied over and over again, would amount to a running time of $\mathcal{O}(3^r \mathrm{poly}(n))$. Having applied it once, we would like to make sure that we will not encounter any further reduced formulas in the subsequent recursive calls. This is too much to ask for, but what we definitely do not want is to encounter a reduced formula $F^{[\gamma]}$ where γ is a pea for F. Think of peas as being especially benign and well-behaved partial assignments. We surely do not want these nice peas to bring us into trouble.

Definition 4.2. *We call a computation of* `searchball`(F, r) good *if for any $F^{[\gamma]}$ occurring in the computation with γ being a pea w.r.t. F and $|\gamma| \geq 1$, the formula $F^{[\gamma]}$ is non-reduced.*

We will give a procedure that runs in reasonable time, and for every reduced F, finds either a satisfying assignment in B_r or a good computation. The benefit of a good computation is clear: As long as our branchings produce peas w.r.t. F, we will not encounter reduced formulas. Recall that Rules 2 and 4 never extend a pea γ to a non-pea. Rules 1 and 3 might, but these rules are so efficient that this compensates for the possibility of encountering a reduced formula afterwards.

We will compute a "guide" formula G for which we find a good computation, and then show that the same computation can be performed on F. Let us be

more precise: We pick a negative clause $\{\bar{x}_1, \bar{x}_2, \bar{x}_3\}$ in $\mathrm{Neg}(F)$, called the *starting clause* of F. For each $i = 1, 2, 3$, we try to extend the partial assignment $[x_i \mapsto 0]$ to a pea γ w.r.t. F such that $F^{[\gamma]}$ is reduced. We do this by recursively applying Rules 1–4, but aborting the recursion on formulas $F^{[\gamma]}$ if γ is not a pea w.r.t. F, or if γ is a pea and $F^{[\gamma]}$ is reduced. This procedure `searchball-prelim` is given in detail in Algorithm 2. The number of leaves visited by `searchball-prelim` is $\leq f_{r-1}$, where

$$f_i := \begin{cases} 2f_{i-1} + 2f_{i-2} & \text{if } i \geq 1, \\ 1 & \text{if } i \leq 0\ . \end{cases} \tag{1}$$

Algorithm 2. `searchball-prelim`(Formula F, partial assignment γ, radius r)

```
1: if γ is not a pea w.r.t. F then
2:    return undefined
3: else if F^[γ] is reduced then
4:    return γ
5: else if r ≤ 0 then
6:    return undefined
7: else
8:    Apply one of Rules 1–4 and obtain a branching {γ_1, ..., γ_ℓ}
9:    for 1 ≤ i ≤ ℓ do
10:      γ' := searchball-prelim(F, γγ_i, r − |γ_i|)
11:      if γ' ≠ undefined then
12:         return γ'
13:      end if
14:   end for
15:   return undefined
16: end if
```

This can easily be seen by induction on r: If Rule 4 is applied, it causes two calls with $r-1$ and two with $r-2$. Rules 1, 2 and 3 are clearly even better. So we call `searchball-prelim`$(F, [x_i \mapsto 0], r-1)$ for each $i = 1, 2, 3$. Suppose each of these three calls returns `undefined`. Then `searchball-prelim` did not encounter any reduced $F^{[\gamma]}$ for γ being a pea, i.e. this was a good computation. Otherwise, let γ_1 be the returned pea and consider $F_2 := F^{[\gamma_1]}$. As for F, pick a negative clause $\{\bar{y}_1, \bar{y}_2, \bar{y}_3\} \in \mathrm{Neg}(F_2)$ as starting clause of F_2 (if there is one). Call `searchball-prelim`$(F_2, [y_i \mapsto 0], r-1)$ for each $i = 1, 2, 3$. Either every call returns `undefined`, or some γ_2 is returned. In the latter case, define $F_3 := F_2^{[\gamma_2]}$ and we do for F_3 what we did for F_2. This creates a sequence $F = F_1, F_2, \ldots$ where $F_{i+1} = F_i^{[\gamma_i]}$, γ_i is a pea w.r.t. F_i returned by `searchball-prelim`, and every F_i is reduced. Since each F_{i+1} contains strictly fewer variables than F_i, this process terminates in some $F_m =: G$. Furthermore, it is not difficult to see that $\gamma_1\gamma_2\ldots\gamma_{m-1}$ is a pea w.r.t. F, and thus $|\gamma_1\gamma_2\ldots\gamma_{m-1}| \leq r$: Recall that F is Neg-disjoint, and we assume $|\mathrm{Neg}(F)| \leq r$, hence every pea has size $\leq r$. See Figure 2.

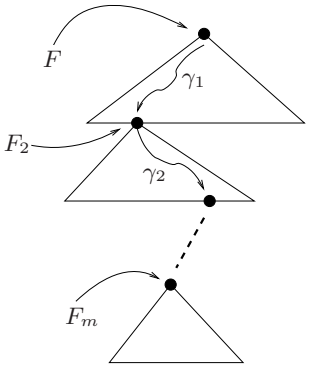

Fig. 2. Constructing a sequence F_1, F_2, ..., F_m of reduced formulas. The process terminates in F_m. Either $\mathrm{Neg}(F_m) = \emptyset$, or for every γ in the tree of F_m that is a pea w.r.t. F_m, $F_m^{[\gamma]}$ is non-reduced.

There are two cases: First, the process above could terminate with $\mathrm{Neg}(G) = \emptyset$. Then setting all variables in G to 1 satisfies it. Since $G = F^{[\gamma_1\gamma_2\cdots\gamma_{m-1}]}$ and $|\gamma_1\gamma_2\ldots\gamma_{m-1}| \leq r$, F is B_r-satisfiable.

Second, the process could terminate with G for which `searchball-prelim` returned `undefined`. Let us contemplate for a moment what this means. When reaching G in the process described above, we pick a starting clause $\{\bar{z}_1, \bar{z}_2, \bar{z}_3\} \in \mathrm{Neg}(G)$ and call `searchball-prelim`$(G, [z_i \mapsto 0], r-1)$ for $i = 1, 2, 3$, and each call returns `undefined`. This means that for any pea γ that occurs in the computation of `searchball-prelim`, $G^{[\gamma]}$ is non-reduced. Therefore, one of Rules 1–4 applies to it, giving a branching Γ. We will show that Γ is valid for $F^{[\gamma]}$, as well, i.e. we can perform the same computation on F instead of G, which will be a good computation of `searchball` on F. We say we use G as a *search guide* for `searchball`(F, r) telling us which branching to apply. Here it is important that the branching in Line 8 of `searchball-prelim` is chosen among all possible branchings by some deterministic rule, such that when performing the same computation for F, we will get exactly the same branching again. This will be a *good* computation, and we will not encounter reduced formulas $F^{[\gamma]}$ as long as γ is a pea w.r.t. G. We need the following technical lemma to show that any branching which is valid for $G^{[\gamma]}$ is also valid for $F^{[\gamma]}$, if γ is a pea w.r.t. G.

Lemma 4.3. *Let F and G be reduced 3-CNFs. Let F_3, G_3 denote the set of all 3-clauses of F and G, respectively. If $G_3 \subseteq F_3$, and γ is a pea w.r.t. G, then* $\mathrm{Neg}(G^{[\gamma]}) \subseteq \mathrm{Neg}(F^{[\gamma]})$.

Proof. Consider any $C \in \mathrm{Neg}(G^{[\gamma]})$. We will show that $C \in \mathrm{Neg}(F^{[\gamma]})$. There is some clause $D \in G$ with $D^{[\gamma]} = C$. If $|D| = 3$, then by assumption $D \in F$ and $C = D^{[\gamma]} \in F^{[\gamma]}$, and we are done. Otherwise, $|D| \leq 2$, and thus D is not a negative clause, because G is reduced. Thus, D is either of the form $\{u\}$, $\{u, \bar{a}\}$ or $\{u, x\}$. If $D = \{u\}$ or $\{u, \bar{a}\}$, then $\gamma(u) = 0$, since $D^{[\gamma]}$ is a negative clause.

Since γ is a pea w.r.t. G, there is a clause $\{\bar{u}, \bar{v}, \bar{w}\}$ in G. Hence Rule 2 or 3 applies to G, contradicting the assumption that G is reduced. If $D = \{u, x\}$, then $C = \emptyset$, and $\gamma(u) = \gamma(x) = 0$. Therefore G contains distinct clauses $\{\bar{u}, \bar{v}, \bar{w}\}$ and $\{\bar{x}, \bar{y}, \bar{z}\}$, and Rule 4 applies. This is again a contradiction. It follows that $|D| = 3$ and $C \in F^{[\gamma]}$, thus $\mathrm{Neg}(G^{[\gamma]}) \subseteq \mathrm{Neg}(F^{[\gamma]})$. □

Lemma 4.4. *Let F and G be reduced 3-CNFs and suppose that $G_3 \subseteq F_3$. If γ is a pea w.r.t. G and one of Rules 1–4 applies to $G^{[\gamma]}$ yielding a branching Γ, then Γ is valid for $F^{[\gamma]}$, as well.*

Proof. Let γ be a pea w.r.t. G. The idea of the proof is the same for each of the four rules, but for the sake of completeness, we will show all four cases.

Case 1. If Rule 1 applies to $G^{[\gamma]}$, then $G^{[\gamma]}$ contains a clause $C = \{\bar{x}_1, \ldots, \bar{x}_\ell\}$ with $\ell \leq 2$. Let $\{[x_1 \mapsto 0], \ldots, [x_\ell \mapsto 0]\}$, $\ell \leq 2$ be the branching. By Lemma 4.3, $C \in F^{[\gamma]}$, thus the branching is valid for $F^{[\gamma]}$.

Case 2. If Rule 2 applies, $G^{[\gamma]}$ contains clauses $C = \{\bar{u}, \bar{v}, \bar{w}\}$ and $D = \{u\}$. By Lemma 4.3, $C \in F^{[\gamma]}$. Note that $\square = D^{[u \mapsto 0]} \in G^{[\gamma, u \mapsto 0]}$. Since $[\gamma, u \mapsto 0]$ is a pea w.r.t. G, too, and we consider the empty clause to be a negative clause, Lemma 4.3 applies again, thus $\square \in F^{[\gamma, u \mapsto 0]}$, and the branching $\{[v \mapsto 0, w \mapsto 0]\}$ is valid for $F^{[\gamma]}$.

Case 3. If Rule 3 applies, there are clauses $\{u, \bar{a}\}$ and $\{\bar{u}, \bar{v}, \bar{w}\}$ in $G^{[\gamma]}$. By Lemma 4.3, $\{\bar{u}, \bar{v}, \bar{w}\} \in F^{[\gamma]}$, as well. Hence the branching $\{[u \mapsto 0], [v \mapsto 0], [w \mapsto 0]\}$ is valid for both $G^{[\gamma]}$ and $F^{[\gamma]}$. Since $[\gamma, u = 0]$ is a pea w.r.t. G, Lemma 4.3 applies and $\{\bar{a}\} \in \mathrm{Neg}(F^{[\gamma, u \mapsto 0]})$. Therefore, the branching $\{[u \mapsto 0, a \mapsto 0], [v \mapsto 0], [w \mapsto 0]\}$ is valid for $F^{[\gamma]}$.

Case 4. If Rule 4 applies, $G^{[\gamma]}$ contains clauses of the form $\{u, x\}$, $\{\bar{x}, \bar{y}, \bar{z}\}$ and $\{\bar{u}, \bar{v}, \bar{w}\}$. By Lemma 4.3, the latter two are in $F^{[\gamma]}$, as well. When applying Rule 4, we do not recurse on $G^{[\gamma, u \mapsto 0, x \mapsto 0]}$, because in this formula, $\{u, x\}$ has become an empty clause. Note that according to our definition, the empty clause is a negative clause, and since $[\gamma, u \mapsto 0, x \mapsto 0]$ is a pea w.r.t. G, Lemma 4.3 implies that $F^{[\gamma, u \mapsto 0, x \mapsto 0]}$ contains the empty clause, too. Therefore the branching $\{[u \mapsto 0, y \mapsto 0], [u \mapsto 0, z \mapsto 0], [v \mapsto 0], [w \mapsto 0]\}$ is valid for $F^{[\gamma]}$. □

Let us summarize our algorithm. If F is not reduced, the algorithm applies one of Rules 1–4. Otherwise, it computes the search guide $G = F^{[\gamma_1 \gamma_2 \cdots \gamma_{m-1}]}$. Let $\{\bar{z}_1, \bar{z}_2, \bar{z}_3\}$ be the starting clause of G. We call `searchball`$(F^{[z_i \mapsto 0]}, r - 1)$ for $i = 1, 2, 3$. As each partial assignment $[z_i \mapsto 0]$ is a pea w.r.t. G, and `searchball-prelim`$(G, [z_i \mapsto 0], r - 1)$ returned `undefined`, $G^{[z_i \mapsto 0]}$ is not reduced, and hence one of Rules 1–4 applies to it, yielding a branching Γ. By Lemma 4.4, this branching is valid for $F^{[z_i \mapsto 0]}$, as well, hence `searchball` applies this very branching in the recursive call `searchball`$(F^{[z_i \mapsto 0]}, r - 1)$. The same argument holds for every subsequent recursive call `searchball`$(F^{[\gamma]}, r')$, as long

as γ is a pea w.r.t. G. If `searchball` is called with some $F^{[\gamma]}$ and γ is not a pea w.r.t. G, we have to discard our search guide. In this case, it may happen that $F^{[\gamma]}$ is reduced again, and we would have to run `searchball-prelim` on $F^{[\gamma]}$, to find a new search guide.

We will now analyze the running time. It turns out that Rule 4 is the "worst case rule" that dominates the running time of the algorithm. However, we have to be careful in our calculations because we do a lot of additional work in `searchball-prelim`. We have to make sure that the running time is still dominated by f_r, defined in (1).

Theorem 4.5. *If F contains no intersecting negative 3-clauses, the number of leaves visited by* `searchball`(F, r) *is* $\leq 3(r+1)^2 f_{r-1}$.

Proof. We prove a stronger statement. We claim in addition that if F is reduced and G is used as a search guide for `searchball`, and γ is a pea w.r.t. G, then `searchball`$(F^{[\gamma]}, r)$ visits $\leq (r+1)^2 f_r$ leaves.

We use induction on r. For $r = 0$, the statement is trivial. If F is reduced, we compute a search guide formula G. Doing this, we call `searchball-prelim` $\leq 3r$ times, each time creating $\leq f_{r-1}$ leaves. Then we pick a clause $\{\bar{y}_1, \bar{y}_2, \bar{y}_3\} \in \mathrm{Neg}(G)$ and call `searchball`$(F^{[y_i \mapsto 0]}, r-1)$ for $i = 1, 2, 3$. Since each $[y_i \mapsto G]$ is a pea w.r.t. G, by induction each call causes $\leq r^2 f_{r-1}$ leaves. Together, this amounts to $\leq 3(r+1)^2 f_{r-1}$ calls.

If F is not reduced and we are not using a search guide, then one of Rules 1–4 applies, and it is straightforward to show that the number of leaves is $\leq 3(r+1)^2 f_{r-1}$.

The most interesting case is when `searchball` is called for $F^{[\gamma]}$, using G as a search guide, and γ is a pea w.r.t. G. If Rule 4 applies, we pick clauses $\{u, x\}$, $\{\bar{x}, \bar{y}, \bar{z}\}$ and $\{\bar{u}, \bar{v}, \bar{w}\} \in G^{[\gamma]}$ and use the branching $\{[u \mapsto 0, y \mapsto 0], [u \mapsto 0, z \mapsto 0], [v \mapsto 0], [w \mapsto 0]\}$. Recall that all four extended assignments are peas w.r.t. G, hence the recursive calls cause $\leq 2r^2 f_{r-1} + 2(r-1)^2 f_{r-2} \leq (r+1)^2 f_r$ leaves. For Rule 2, the argument is exactly the same. This is really the crucial point in this algorithm: Of all four rules, Rule 4 yields the worst running time. However, using our search guide, we can be sure not to encounter a reduced formula after applying Rule 4.

If Rule 3 applies, we pick $\{u, \bar{a}\}, \{\bar{u}, \bar{v}, \bar{w}\} \in G^{[\gamma]}$ and use the branching $\{[u \mapsto 0, a \mapsto 0], [v \mapsto 0], [w \mapsto 0]\}$. Note that $\gamma[v \mapsto 0]$ and $\gamma[w \mapsto 0]$ are peas w.r.t. G, hence these calls cause $\leq 2r^2 f_{r-1}$ leaves. However, $\gamma[u \mapsto 0, a \mapsto 0]$ is perhaps not a pea w.r.t. G, hence $F^{[u \mapsto 0, a \mapsto 0]}$ might be reduced, and this call causes $\leq 3(r-1)^2 f_{r-3}$ leaves. Altogether, this is surely $\leq 2(r+1)^2 f_{r-1} + 3(r+1)^2 f_{f-3}$, which is $\leq (r+1)^2 f_r$. If Rule 1 applies, we cause $\leq 2 \times 3r^2 f_{r-2} \leq 3(r+1)^2 f_{r-1}$ leaves. This completes the proof. □

To summarize, computing and using a search guide guarantees that reduced formulas might be encountered once, but in subsequent calls, they will be encountered only after Rule 1 or Rule 3 has been applied. These rules are so efficient that they compensate for the possibility of encountering a reduced formula

afterwards. To complete our algorithm, we have to show finally what to do when F is not Neg-disjoint. We basically do the same as Brueggemann and Kern [2].

Theorem 4.6. *Let F be a 3-CNF. The number of leaves visited by* `searchball`(F, r) *is* $\leq 3(r+1)^2 f_{r-1}$.

Proof. If F contains a negative clause of size ≤ 2, we obtain by induction that `searchball`(F, r) causes $\leq 2 \times 3r^2 f_{r-2} \leq 3(r+1)^2 f_{r-1}$ leaves. Otherwise, all negative clauses are of size three. There are three cases:

First, there could be clauses $\{\bar{u}, \bar{v}, \bar{w}\}$, $\{\bar{u}, \bar{v}, \bar{z}\}$ intersecting in exactly two literals. We use the branching $\{[u \mapsto 0], [u \mapsto 1, v \mapsto 0], [u \mapsto 1, v \mapsto 1, w \mapsto 0, z \mapsto 0]\}$. Though not valid according to our definition, it still holds that F is B_r-satisfiable if and only if $F^{[\gamma]}$ is $B_{r-|\gamma|}$-satisfiable, for some γ in the branching. The claimed time bound follows after a short computation.

Second, if F contains two 3-clauses $\{\bar{u}, \bar{v}, \bar{w}\}$, $\{\bar{u}, \bar{y}, \bar{z}\}$ intersecting in exactly one literal, use the branching $\{[u \mapsto 0], [u \mapsto 1, v \mapsto 0, y \mapsto 0], [u \mapsto 1, v \mapsto 0, z \mapsto 0], [u \mapsto 1, w \mapsto 0, y \mapsto 0], [u \mapsto 1, w \mapsto 0, z \mapsto 0]\}$. Again, a short calculation shows that this causes $\leq 3(r+1)^2 f_{r-1}$ leaves.

Third, it could be that F does not contain intersecting negative clauses. Then Theorem 4.5 applies. This completes the proof. □

It is standard to show that $f_r \in \mathcal{O}(a^r)$, where $a \approx 2.74$ is the largest root of $x^2 - 2x - 2$. Therefore, B_r-satisfiability can be decided in time $\mathcal{O}(a^r \mathrm{poly}(n))$, and thus, by the results of Danstin et al., we can decide 3-SAT in time $\mathcal{O}\left(1.465^n \mathrm{poly}(n)\right)$.

5 Conclusions

Observe that Lemma 4.3, though looking innocent, is really the core reason why our algorithm works. It is also what causes trouble when ones tries to directly apply guided search to other backtracking algorithms. Take for example Beigel and Eppstein's Algorithm [1] for solving (3,2)-CSP. For this algorithm we cannot find and apply an equivalent of Lemma 4.3, because the algorithm uses some kind of resolution which introduces new constraints, whereas our application of Lemma 4.3 relies crucially on the fact that if G was created from F by steps of the algorithm, then G does not contain any 3-clauses that F does not contain.

For traditional backtracking algorithms for k-SAT, often called *DPLL algorithms*, after Davis, Putnam, Logemann and Loveland [4,5], there is an even simpler technique than guided search. Consider a backtracking algorithm that chooses a shortest clause $C = \{u_1, \ldots, u_i\} \in F$ and recurses on $F_1 := F^{[u_1 \mapsto 1]}$ and $F_0 := F^{[u_1 \mapsto 0]}$, where F_0 contains an $(i-1)$-clause. In this context, call a formula *reduced* if every clause in F has size exaclty k. It is clear that if F is a k-CNF and in the recursive computation of F, a reduced formula $F^{[\gamma]}$ occurs, then $F^{[\gamma]} \subset F$, and the two formulas are *SAT-equivalent* in the sense that one is satisfiable if and only if the other is. Hence all other open branches of the search tree can be pruned, and the algorithm only needs to recurse on $F^{[\gamma]}$. This is known

as the *autark assignment rule* and was used by Monien and Speckenmeyer [6] to speed up their k-SAT algorithm.

It should be mentioned that we first tried to prove some kind of autarky result, i.e. that if F is reduced and $F^{[\gamma]}$ is reduced as well, then one formula is B_r-satisfiable if and only if the other is. Unfortunately, this is not true. One can, however, obtain a SAT-equivalence under certain conditions stronger than reducedness, which leads to a simpler proof of the $\mathcal{O}(1.473^n)$-bound of Brueggemann and Kern [2].

Acknowledgements

I want to thank Andreas Razen and Emo Welzl for fruitful and thorough discussions, which greatly helped to improve this paper. Further I am grateful for the comments and suggestions of the referees.

References

1. Beigel, R., Eppstein, D.: 3-coloring in time $O(1.3446^n)$: a no-MIS algorithm. In: Proc. 36th Symp. Foundations of Computer Science, October 1995, pp. 444–453. IEEE, Los Alamitos (1995)
2. Brueggemann, T., Kern, W.: An improved local search algorithm for 3-SAT. Memorandum 1709, Department of Applied Mathematics, University of Twente, Enschede (2004)
3. Dantsin, E., Goerdt, A., Hirsch, E.A., Kannan, R., Kleinberg, J., Papadimitriou, C., Raghavan, O., Schöning, U.: A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. Theoretical Computer Science 289, 69–83 (2002)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Comm. ACM 5, 394–397 (1962)
5. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. Assoc. Comput. Mach. 7, 201–215 (1960)
6. Monien, B., Speckenmeyer, E.: Solving satisfiability in less than 2^n steps. Discrete Applied Mathematics 10, 287–295 (1985)
7. Rolf, D.: Improved bound for the PPSZ/Schöning-algorithm for 3-SAT. Electronic Colloquium on Computational Complexity (ECCC) 159 (2005)