Handbook of Satisfiability
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
IOS Press, 2009

Chapter 7

Fundaments of Branching Heuristics

Oliver Kullmann

7.1. Introduction

The topic of this chapter is to provide foundations for “branching heuristics”. The whole field of “heuristics” is very diverse, and we will concentrate on a specific part, where developments in the last four decades can be comprised in what actually deserves to be called a “theory”. A full version of this chapter, containing all proofs and extensive examples, is available in [Kul08a].

The notion of a “heuristics” is fundamental for the field of Artificial Intelligence. The (early) history of the notion of “heuristics” is discussed in [BF81], Section II.A, and the usage of this notion in the SAT literature follows their definition of a heuristics as a method using additional information to restrict the search space size, though in this chapter we consider a restricted context, where completeness is not an issue, but the heuristical component of the search process affects only resource usage, not correctness. Furthermore we only study a specific form of search processes here, namely backtracking search. We consider the situation where we have a problem instance F where all direct (“efficient”) methods fail, and so F has to be split into subproblems. In the context of this chapter we basically assume that the method for splitting F is already given, yielding possible “branchings” $F \rightsquigarrow F_1, \dots, F_m$, splitting F into m “subproblems” F_i, and the task of the heuristic is to compare different branchings and to find the “best” branching among them. Let us assume that we have given three branchings to compare:

$$\begin{tikzcd}
 & F \arrow[dl, "2"'] \arrow[d, "2"] \\
F^1_1 & F^1_2
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & F \arrow[dll, "3"'] \arrow[dl, "3"] \arrow[d, "4"] \arrow[dr, "9"] & \\
F^2_1 & F^2_2 & F^2_3 & F^2_4
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & & F \arrow[dlll, "4"'] \arrow[dll, "5"] \arrow[dl, "6"] \arrow[d, "6"'] \arrow[dr, "5"] \arrow[drr, "6"] & & \\
F^3_1 & F^3_2 & F^3_3 & F^3_4 & F^3_5 & F^3_6
\end{tikzcd}$$

We see an important aspect of our abstract analysis: Each branch is labelled by a positive real number, which measures the “distance”, that is, how much “simpler” the problem became (w.r.t. a certain aspect). At this level of abstraction not only the branchings but also these distances are given, and the question is what can be done with these numbers, how can they be used to compare these three branchings? Obviously, all that counts then are the tuples of distances, so-called "branching tuples". We get $a := (2, 2)$, $b := (3, 3, 4, 9)$, $c := (4, 5, 6, 6, 5, 6)$.

The first part of this chapter is devoted to the combinatorics and analysis of branching tuples. We will see that a canonical value $\tau(t) \in \mathbb{R}_{\geq 1}$ (see Definition 7.3.2) can be computed for each branching tuple, and that these values under fairly general circumstances yield the (only) answer to the problem of comparing branching tuples: The smaller the τ-value the better the tuple. For the above branching tuples the computation yields $\tau(a) = 1.4142\ldots$, $\tau(b) = 1.4147\ldots$, and $\tau(c) = 1.4082\ldots$, and thus the third branching is the best, followed by the first branching, and then the second branching (all under the assumption that all that is given are these numbers). The τ-function arises naturally from standard techniques from difference equations, and for example $\tau(b)$ is the (unique) positive solution of $x^{-3}+x^{-3}+x^{-4}+x^{-9} = 1$. Using an appropriate scaling, the τ-function yields a (generalised) mean $\mathfrak{T}$ (in a precise sense), and the task of evaluating branchings can be understood as extracting a generalised mean-value from each tuple (where in this interpretation the tuple is the better the larger the mean). It is worth mentioning here that branching tuples are arbitrary tuples of positive *real* numbers, and thus are amenable to optimisation techniques.

After having developed a reasonable understanding of branching tuples, based on the analysis of *rooted trees* which consist of these branching tuples w.r.t. a run of the given backtracking solver, we then turn to the question of how actually to compute "good" distances. At this time the general theory can only give rough guidelines, which however suffices to *compare* different distance functions w.r.t. their appropriateness. So we can compare distance functions, and we can also optimise them. But finally we have to come up with some concrete distance, and fortunately, in the second part of this chapter, at least for CNF in the context of so-called "look-ahead" solvers a reasonable answer can be given, obtained by a remarkable convergence of theoretical and empirical studies. We need to remark here that heuristics for so-called "conflict-driven" solvers cannot be fully handled here, basically since these solvers are not really based on backtracking anymore, that is, we cannot speak of *independent* branches anymore, but rather an iterative process is taking place.

In more details, the content of this chapter is as follows:

1. The general framework and its assumptions are discussed in Section 7.2. First we discuss the focus of our approach, and which questions are excluded by this theory, which is based on comparing branchings by condensing them into branching tuples. The way these tuples arise is based on "measures" (of approximated problem complexity) or more generally on "distances", and we discuss the meaning of this process of extracting branching tuples. Alternative branchings are represented by alternative branching tuples, and so we need to order branching tuples by some linear quasi-order (smaller is better), choosing then a branching with smallest associated branching tuple. This order on the set of branching tuples is given by applying a "projection function", and by an introductory example

we motivate the "canonical projection", the τ-function.

2. The combinatorics of branching tuples and the fundamental properties of the τ-function are the subject of Section 7.3. In particular we consider bounds on the τ-function (resp. on the associated mean $\mathfrak{T}$). These bounds are important to derive upper and lower bounds on tree sizes. And under special circumstances one might wish to consider alternatives to the τ-function as a "projection" (comprising a branching tuple to one number), and then these bounds provide first alternatives. Section 7.3 is concluded by introducing the fundamental association of probabilities with branching tuples.
3. Then in Section 7.4 we discuss the fundamental method of estimating tree sizes, first for trees with given probability distributions, and then for trees where the probability distribution is derived from a given distance by means of the τ-function.
4. The τ-function is "in general" the only way to comprise branching tuples into a single number, however the precise value of the τ-function is not of importance, only the linear quasi-order it induces on the set of all branching tuples; this is proven in Section 7.5.
5. Though the τ-function is the canonical projection in general, there might be reasons to deviate from it under special circumstances. The known facts are presented in Section 7.6. For binary branching tuples (t_1, t_2) (dominant for practical SAT solving) we relate the projections $t_1 + t_2$ and $t_1 \cdot t_2$ to the τ-function, yielding a strong analytical argument why the "product rule" is better than the "sum rule", complementing the experimental evidence.
6. With Section 7.7 we enter the second part of this chapter, and we discuss the known distances w.r.t. practical SAT solving.
7. As already mentioned, at present there is no theory for choosing a "good" distance for a particular class of problem instances (other than the trivial choice of the optimal distance), but from the general theory developed in the first part of this chapter we obtain methods for improving distance functions, and this is discussed in Section 7.8.
8. Our approach on branching is based on a two-phase model, where first the branching itself is chosen, and then, in a second step, the order of the branches. Methods for finding good orders (for SAT problems) are discussed in Section 7.9.
9. The ideas for concrete distances, as presented in Section 7.7, also have bearings on more general situations than just boolean CNF-SAT, and especially our general theory is applicable in a much larger context. A quick outlook on this topic is given in Section 7.10.

7.2. A general framework for branching algorithms

A general framework for heuristics for branching algorithms is as follows: Consider a non-deterministic machine $\mathcal{M}$ for solving some problem, where the computation terminates and is correct on *each* possible branch, and thus the decisions made during the run of the machine only influence resource consumption. The task for a heuristics $\mathcal{H}$ is to make the machine deterministic, that is, at each choice point to choose one of the possible branches, obtaining a deterministic machine $\mathcal{M}_{\mathcal{H}}$, where typically time and/or space usage is to be minimised. Likely not much can be said about the choice-problem in this generality, since no information is given about the choices. The focus of this article is on the problem of good choices between different possibilities of splitting problems into (similar) *subproblems*, where for each possible choice (i.e., for each possible splitting) we have (reasonable) information about the subproblems created. Not all relevant information usable to gauge branching processes for SAT solving can be represented (well) in this way, for example non-local information is hard to integrate into this "recursive" picture, but we consider the splitting-information as the central piece, while other aspects are treated as "add-ons".

7.2.1. Evaluating branchings

The basic scenario is that at the current node v of the backtracking tree we have a selection $\mathcal{B}(v) = (B_1, \ldots, B_m)$ of branchings given, and the heuristic chooses one. Each branching is (in this abstract framework) considered as a tuple $B_i = (b_1, \ldots, b_k)$ of branches, where each b_i is a "smaller" problem instance, and k is the width of the branching. If the determination of the order of branches is part of the heuristics, then all $k!$ permutations of a branching are included in the list $\mathcal{B}(v)$, otherwise a standard ordering of branches is chosen. If we consider branching on a boolean variable, where the problem instance contains n variables, and all of them are considered by the heuristics, then the selection $\mathcal{B}(v)$ contains $2n$ branchings if the order of the two branches is taken into account, and only n branchings otherwise.

The problem is solved in principle, if we have precise (or "good") knowledge about the resource consumption of the subproblems b_i (in the order they are processed, where the running time of b_i might depend on the running time of b_j for $j < i$), since then for every possible branching we sum up the running times of the branches (which might be 0 if the branch is not executed) to get the total running time for this branching, and we choose a branching with minimal running time. If ordinary backtracking order is not followed (e.g., using restarts, or some form of evaluating the backtracking tree in some other order), or branches influence other branches (e.g., due to learning), then this might be included in this picture by the assumption of "complete clairvoyance".

Though this picture is appealing, I am not aware of any circumstances in (general) SAT solving where we actually have good enough knowledge about the resource consumption of the subproblems b_i to apply this approach successfully. Even in the probabilistically well-defined and rather restricted setting of random 3-SAT problems, a considerable effort in [Ouy99] (Chapter 5) to construct such a "rational branching rule" did not yield the expected results. The first step towards a practical solution is to use (rough) estimates of problem complexity, captured by a measure $\mu(F)$ of "problem complexity". We view $\mu(F)$ as a kind of logarithm of the true complexity. For example, the trivial SAT algorithm has the bound $2^{n(F)}$, and taking the logarithm (base 2) we obtain the most basic measure of problem complexity here, the number $n(F)$ of variables. This "logarithmic" point of view is motivated by the optimality result Lemma 7.4.9. Progress in one branch $F \rightsquigarrow F'$ then can be measured by $\Delta\mu(F, F') = \mu(F) - \mu(F') > 0$. However, since at this time the practical measures $\mu(F)$ are too rough for good results, instead of the difference $\Delta\mu(F, F')$ a more general "distance" $d(F, F') > 0$ needs to be involved, which estimates, in some heuristic way, the prospects F' offers to actually be useful in the (near) future (relative to F, for the special methods of the algorithm considered). Before outlining the framework for this kind of analysis, two basic assumptions need to be discussed:

- A basic assumption about the estimation of branching quality is *homogeneity*: The branching situation might occur, appropriately "relocated", at many other places in the search tree, and is not just a "one-off" situation. If we have a "special situation" at hand (and we are aware of it), then, in this theoretical framework, handling of this special situation is not the task of the heuristics, but of the "reduction" for the currently given problem (compare Subsection 7.7.2).
- Another abstraction is, that as for theoretical upper-bounds, a "mathematical running time" is considered: The essential abstraction is given by the *search tree*, where we ignore what happens "inside a node", and then the mathematical running time is the number of nodes in this tree. Real running times (on real machines) are not considered by (current, abstract) heuristics. In the literature (for example in [Ouy99]) one finds the attempt of taking the different workloads at different nodes into account by measuring the (total) number of assignments to variables, but this works only for special situations, and cannot be used in general. Furthermore, practice shows that typically, given the number of nodes in the search tree and some basic parameters about the problem size, curve-fitting methods yield good results on predicting the actual running time of a solver.[1]

7.2.2. Enumeration trees

We focus on algorithms where an enumeration tree is built, and where the main heuristical problem is how to control the growth of the tree.

1. We do not consider "restarts" here, that is, rebuilding the tree, possibly learning from prior experiences. In the previous phase of SAT-usage, emphasise was put on *random* restarts, with the aim of undoing bad choices, while in the current phase randomness seems no longer important, but the effects of *learning* are emphasised. In this sense, as remarked by John Franco, restarts can be seen as a kind of "look-ahead".

[1]The framework we develop here actually is able to handle different run times at different nodes by attaching different weights to nodes. However in this chapter, where the emphasise is on setting up the basic framework, we do not consider this; see [Kul08a] for how to generalise the framework.

2. We assume mostly a depth-first strategy.
3. "Intelligent backtracking" (that is, not investigating the second branch in the case of an unsatisfiable first branch where the conflict does not depend on the branching) is considered as accidental (not predictable), and thus not included in heuristical considerations.
4. "Jumping around" in the search tree in order to prioritise nodes with higher probability of finding a satisfying assignment (like in [Hv08]) is considered as a layer on top of the heuristics (a kind of control layer), and is not considered by the core heuristics (given the current state of research).
5. The effect one branch might have on others (via "global learning") is also considered a secondary effect, out of our control and thus not directly covered by the heuristics.
6. As discussed in [Kul08b], conflict-driven solvers actually tend to choose an iterative approach, not creating independent sub-problems for branching, but just choosing one branch and leaving it to learning to cater for completeness. Such a process seems currently unanalysable.

After having outlined what is to be included in our analysis, and what is to be left out, we now give a first sketch of the framework to be developed in this article.

7.2.3. A theoretical framework

For the author of this chapter, the theory developed here originated in the theoretical considerations for proving upper bounds on SAT decision (see Section 7.7.3 for further comments), and so the underlying "driving force" of the heuristics is the minimisation of an upper bound on the search tree size. Only later did I see that in [Knu75] actually the same situation is considered, only from a different point of view, a probabilistic one where a path through the search tree is chosen by randomly selecting one successor at each inner node.[2]

Example 7.2.1. An instructive example is the first non-trivial 3-SAT bound ([Luc84, MS85]), where via the autarky argument it can be assumed that a binary clause is always present, and then splitting on a variable in this binary clauses eliminates one variable in one branch and two variables in the other (due to unit-clause elimination). "Traditionally" this is handled by considering the worst case and using a difference equation $f_n = f_{n-1} + f_{n-2}$ (n is the number of variables, while f_n is the number of leaves in the search tree). Reasonably we assume $f_0 = 0$ and $f_1 = 1$, and then $(f_n)_{n \in \mathbb{N}_0}$ is the Fibonacci sequence with $f_n = \frac{1}{\sqrt{5}}\big((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n\big)$, and using $r : \mathbb{R} \setminus (\frac{1}{2} + \mathbb{Z}) \to \mathbb{Z}$ for rounding to the nearest integer, we have $f_n = r(\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n)$ (see Chapter 1 in [CFR05] for an introduction into these considerations). Whatever the initial values f_0, f_1 are, we always have $f_n = \Theta((\frac{1+\sqrt{5}}{2})^n) = O(1.619^n)$. The fundamental approach is

[2]In fact, the proof of the central "τ-lemma" in [Kul99b] used already the probabilistic interpretation, but [Knu75] was not known to me at this time.

the Ansatz $f_n = \lambda^n$, which leads to the equation $\lambda^n = \lambda^{n-1} + \lambda^{n-2}$, which is equivalent to $\lambda^2 = \lambda + 1$.

We do not follow the general theory of difference equations any further, since the initial conditions are of no relevance, and we also want to allow descents in arbitrary positive *real* numbers, in order to allow optimisations. The computation of the root λ from Example 7.2.1 will be generalised in Definition 7.3.2, while Theorem 7.4.8 yields a general method for computing bounds on tree sizes. Regarding our main purpose, the evaluation of branchings, we can show (in Section 7.5) that this generalised λ-calculation is the only "general" way of projecting a branching tuple to a single number.

As we have already mentioned, the first step is to move from branchings to branching tuples. For Example 7.2.1 this means extracting the branching tuple $(1, 2)$ as the essential piece of information. Now an important aspect of the theory developed in this chapter is that branching tuples are not only considered in isolation, but in connection with the (search) trees (in abstracted form). This "emancipation" of search trees, which in the form of worst-case analysis based on recurrence equations are factually suppressed, is also important for practical applications, as demonstrated in Section 7.8, since it allows to consider the real computations going on, not just their shadows in the form of worst-case considerations, which must deliver some interpretable bound at the end. And furthermore the study of probability distributions on the leaves of the tree combine the worst-case upper bounds with the probabilistic considerations from [Knu75] — the λ-calculation associates, in a canonical way, a probability distribution to the branches of a branching tuple (see Subsection 7.3.4).

So a main part of the theory is concerned with estimating sizes of trees in connection with branching tuples associated with them, based on simple probability considerations. For this task, the above λ-calculation takes on the more general form of evaluating a branching tuple by a real number. We call such evaluations *evaluation projection*, but actually, since the only form of projections used in this paper are such evaluation projections, we just call them "projections". The general study of projections yields the underlying theory to answer questions like "Why is the product-rule for SAT heuristics better than the sum-rule?" (see Section 7.6). We begin the development of the theoretical framework by taking a closer look at branching tuples and the "λ-calculation".

7.3. Branching tuples and the canonical projection

The subject of this section is the theory of "branching tuples" and their evaluation. The theory shows its full strength when considering branchings of arbitrary width, which at this time is of importance for theoretical upper bounds (see Section 7.7.3) and for constraint satisfaction (see Section 7.10.2), and which might become more important for practical SAT solving when for example considering deeper look-aheads at the root of the search tree.

7.3.1. Branching tuples and their operations

One single (potential) branching consisting of $k \in \mathbb{N}$ branches is evaluated by a "branching tuple" a of length k, attaching to each branch i a positive real number a_i, which is intended to measure the progress achieved in reducing problem complexity in this branch (thus the larger a_i, the better is branch i).

Definition 7.3.1. $\boldsymbol{\mathcal{BT}} := \bigcup_{k \in \mathbb{N}} (\mathbb{R}_{>0})^k$ denotes the set of **branching tuples**.

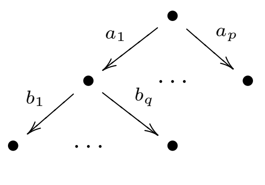
Remarks:

1. Basic measurements for branching tuples are minimum $\min : \mathcal{BT} \to \mathbb{R}_{>0}$, maximum $\max : \mathcal{BT} \to \mathbb{R}_{>0}$, sum $\Sigma : \mathcal{BT} \to \mathbb{R}_{>0}$, and width $|| : \mathcal{BT} \to \mathbb{N}$. The minimum $\min(a)$ of a branching tuple is a "worst-case view", while $\max(a)$ is a "best-case view". In general, disregarding the values, the larger $|a|$, i.e., the wider the branching is, the worse it is.
2. The set of branching tuples of width k is $\boldsymbol{\mathcal{BT}^{(k)}} := \{t \in \mathcal{BT} : |t| = k\}$, which is a *cone*, that is for $a \in \mathcal{BT}^{(k)}$ and $\lambda \in \mathbb{R}_{>0}$ we have $\lambda \cdot t \in \mathcal{BT}^{(k)}$, and for $a, b \in \mathcal{BT}^{(k)}$ we have $a + b \in \mathcal{BT}^{(k)}$.
3. Branching tuples of width 1, which do not represent "real branchings" but "reductions", are convenient to allow.
4. One could also allow the empty branching tuple as well as the zero branching tuple (0) (of width 1), but for the sake of simplicity we abstain from such systematic extensions here.

Concatenation of branching tuples a, b is denoted by "$a\,;b$", and yields the semigroup $(\mathcal{BT}, ;)$ (the empty branching tuple would be the neutral element here). The width function $||$ now becomes a homomorphism from $(\mathcal{BT}, ;)$ to $(\mathbb{N}, +)$. Concatenation allows us to define the strict prefix order $a \sqsubset b :\Leftrightarrow \exists\, x \in \mathcal{BT} : a\,;x = b$ (that is, b is obtained from a by adding further positive numbers to the end of a) for $a, b \in \mathcal{BT}$, while $a \sqsubseteq b :\Leftrightarrow a \sqsubset b \vee a = b$. A further basic operation for a branching tuple a of width k is to apply a *permutation* $\pi \in S_k$, which we denote by $\pi * a := (a_{\pi(1)}, \ldots, a_{\pi(k)})$. Finally we have **composition of branching tuples** a, b at position i of a, that is, branching b is attached to branch i of a; since we allow permutation of branching tuples, it suffices to set $i = 1$ here, and the resulting composition is denoted by $\boldsymbol{a} \mathbin{/\!\!\backslash\!\!\backslash} \boldsymbol{b}$, defined as

$$(a_1, \ldots, a_p) \mathbin{/\!\!\backslash\!\!\backslash} (b_1, \ldots, b_q) := (a_1 + b_1, \ldots, a_1 + b_q, a_2, \ldots, a_p)$$

$=$



We obtain a semigroup $(\mathcal{BT}, \mathbin{/\!\!\backslash\!\!\backslash})$ (non-commutative; the zero branching tuple (0) would be the neutral element). It is important to realise that it makes a difference where to attach the branching b, that is in our setting, the branching tuples $a \mathbin{/\!\!\backslash\!\!\backslash} b$ and $(\pi * a) \mathbin{/\!\!\backslash\!\!\backslash} b$ are in general qualitatively different:

(i) if b is better than a, then attaching b to a smaller component of a yields a better tuple than when attaching it to a larger component;
(ii) if b is worse than a, then attaching b to a smaller component of a yields a worse tuple than when attaching it to a larger component.

The intuitive reason is that "more balanced tuples are better", and so in the first case it is a greater improvement to a when improving its weak parts than when improving its strong parts, while in the second case making a weak part worse means a greater impairment than making a strong part worse.

7.3.2. The tau-function

In this section we introduce formally the τ-function as discussed in Example 7.2.1, and show its main properties.

Definition 7.3.2. Define $\chi_k : \mathcal{BT} \times \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ by $\chi(t,x) := \sum_{i=1}^{|t|} x^{-t_i}$. Observe that for each $t \in \mathcal{BT}$ the map $\chi(t,-) : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ is strictly decreasing with $\chi(t,1) = |t| \geq 1$ and $\lim_{x\to\infty} \chi(t,x) = 0$. Now $\tau : \mathcal{BT} \to \mathbb{R}_{\geq 1}$ is defined as the unique $\tau(t) := x_0 \in \mathbb{R}_{\geq 1}$ such that $\chi(t)(x_0) = 1$ holds.

By definition we have $\tau(t) \geq 1$, with $\tau(t) = 1 \Leftrightarrow |t| = 1$. For $k \in \mathbb{N}$ we denote by $\tau_k : \mathcal{BT}^{(k)} \to \mathbb{R}_{\geq 1}$ the τ-function restricted to branching tuples of width k.

Example 7.3.1. The computation of $\tau(1,2)$ (recall Example 7.2.1) leads to the equation $x^{-1} + x^{-2} = 1$, which is equivalent to $x^2 - x - 1 = 0$, which has the two solutions $\frac{1\pm\sqrt{5}}{2}$, and thus $\tau(1,2) = \frac{1+\sqrt{5}}{2} = 1.6180\ldots$ Only very few τ-values can be expressed by analytical formulas, and most of the time numerical computations have to be used (see Remark 2 to Corollary 7.3.5), so for example $\tau(2,3,5) = 1.4291\ldots$

Simple properties can be proven directly:

Lemma 7.3.1. *For every $a \in \mathcal{BT}$, $k \in \mathbb{N}$ and $\lambda \in \mathbb{R}_{>0}$ we have:*

1. $\tau(\lambda \cdot a) = \tau(a)^{1/\lambda}$.
2. $\tau_k(\vec{1}) = k$.
3. τ_k *for $k \geq 2$ is strictly decreasing in each component.*
4. τ_k *is symmetric, that is, invariant under permutation of branching tuples.*
5. $\tau(a)^{\min(a)} \leq |a| \leq \tau(a)^{\max(a)}$*, that is,* $|a|^{1/\max(a)} \leq \tau(a) \leq |a|^{1/\min(a)}$.
6. $\lim_{\lambda\to 0} \tau(a;(\lambda)) = \infty$ *and* $\lim_{\lambda\to\infty} \tau(a;(\lambda)) = \tau(a)$.

The τ-function fulfils powerful convexity properties, from which non-trivial further properties will follow. A function $f : C \to \mathbb{R}$ defined on some convex subset $C \subseteq \mathbb{R}^k$ is called "strictly convex" if for all $x, y \in C$ and $0 < \lambda < 1$ holds $f(\lambda x + (1-\lambda)y) < \lambda f(x) + (1-\lambda) f(y)$; furthermore f is called "strictly concave" if $-f$ is strictly convex. By definition τ_1 is just the constant function with value 1, and so doesn't need to be considered here.

Lemma 7.3.2. *For $k \geq 2$ the function τ_k is strictly convex.*

Lemma 7.3.2 strengthens considerably the quasi-convexity of τ_k as shown in Lemma 4.1 of [Epp06]; in Corollary 7.3.5 we shall prove a further strengthening.

7.3.3. Bounds on the tau-function

Just from being symmetric and strictly convex it follows, that $\tau_k(a)$ for tuples a with a given fixed sum $\Sigma(a) = s$ attains its strict minimum for the constant tuple (with entries $\frac{s}{k}$); see Lemma 7.3.3 below for a proof. Thus, using $\mathfrak{A}(t) := \sum(t)/|t|$ for the arithmetic mean of a branching tuple, we have $\tau(\mathfrak{A}(t) \cdot \vec{1}) \le \tau(t)$ (with strict inequality iff t is not constant). In the remainder of this subsection we will be concerned with further estimations on the τ-functions, and for that purpose we use the following well-known means (see [HLP99, Bul03]):

1. The *arithmetic mean*, the *geometric mean*, and the *harmonic mean* of branching tuples t are denoted respectively by $\mathfrak{A}(t) := \frac{1}{|t|}\Sigma(t) = \frac{1}{|t|}\sum_{i=1}^{|t|} t_i$, $\mathfrak{G}(t) := \sqrt[|t|]{\prod_{i=1}^{n} t_i}$ and $\mathfrak{H}(t) := |t|/\big(\sum_{i=1}^{n} \frac{1}{t_i}\big)$. We recall the well-known fundamental inequality between these means: $\mathfrak{H}(t) \le \mathfrak{G}(t) \le \mathfrak{A}(t)$ (where strict inequalities hold iff t is not constant).
2. More generally we have the *power means* for $\alpha \in \overline{\mathbb{R}}$ given by $\mathfrak{M}_\alpha(t) := \big(\frac{1}{|t|}\sum_{i=1}^{|t|} t_i^\alpha\big)^{1/\alpha}$ for $\alpha \notin \{-\infty, 0, +\infty\}$, while we set $\mathfrak{M}_{-\infty}(t) := \min(t)$, $\mathfrak{M}_0(t) := \mathfrak{G}(t)$ and $\mathfrak{M}_{+\infty}(t) := \max(t)$. By definition we have $\mathfrak{M}_{-1}(t) = \mathfrak{H}(t)$ and $\mathfrak{M}_1(t) = \mathfrak{A}(t)$. In generalisation of the above fundamental inequality we have for $\alpha, \alpha' \in \overline{\mathbb{R}}$ with $\alpha < \alpha'$ the inequality $\mathfrak{M}_\alpha(t) \le \mathfrak{M}_{\alpha'}(t)$, which is strict iff t is not constant.

We want to establish a variation of the τ-function as a "mean", comparable to the above means. In the literature there are no standard axiomatic notions about "means", only collections of relevant properties, and we condense the relevant notion here as follows:

Definition 7.3.3. Consider $k \in \mathbb{N}$. A **mean** is a map $M : \mathcal{BT}^{(k)} \to \mathbb{R}_{>0}$ which is continuous, strictly monotonic increasing in each coordinate, symmetric (i.e., invariant under permutation of the arguments) and "consistent", that is, $\min(a) \le M(a) \le \max(a)$ for $a \in \mathcal{BT}^{(k)}$. A mean M is **homogeneous** if M is positive homogeneous, i.e., for $\lambda \in \mathbb{R}_{>0}$ and $a \in \mathcal{BT}^{(k)}$ we have $M(\lambda \cdot a) = \lambda \cdot M(a)$.

All power means are homogeneous means. Yet k-ary means M are only defined for tuples of positive real numbers $a \in \mathbb{R}_{>0}^k$, and we extend this as follows to allow arguments 0 or $+\infty$, using the extended real line $\overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$. We say that M is defined for $a \in \overline{\mathbb{R}}_{\ge 0}^k$ (allowing positions to be 0 or $+\infty$) if the limit $\lim_{a' \to a, a' \in \mathbb{R}_{>0}^k} M(a')$ exists in $\overline{\mathbb{R}}$, and we denote this limit by $M(a)$ (if the limit exists, then it is unique). Power means $\mathfrak{M}_\lambda(a)$ for $\lambda \ne 0$ are defined for all $a \in \overline{\mathbb{R}}_{\ge 0}^k$, while $\mathfrak{M}_0(a) = \mathfrak{G}(a)$ is defined iff there are no indices i, j with $a_i = 0$ and $a_j = \infty$.

Definition 7.3.4. Consider a mean $M : \mathcal{BT}^{(k)} \to \mathbb{R}_{>0}$. We say that M is ∞-**dominated** resp. 0-**dominated** if for every $a \in \overline{\mathbb{R}}_{\ge 0}^k$, such that an index i with $a_i = \infty$ resp. $a_i = 0$ exists, $M(a)$ is defined with $M(a) = \infty$ resp. $M(a) = 0$. On the other hand, M **ignores** ∞ resp. **ignores** 0 if $M(a; (\infty))$ resp. $M(a; (0))$ is defined iff $M(a)$ is defined with $M(a; (\infty)) = M(a)$ resp. $M(a; (0)) = M(a)$.

Power means $\mathfrak{M}_\lambda$ with $\lambda > 0$ are ∞-dominated and ignore 0, while for $\lambda < 0$ they are 0-dominated and ignore ∞. The borderline case $\mathfrak{M}_0 = \mathfrak{G}$ is ∞-dominated as well as 0-dominated if only tuples are considered for which $\mathfrak{G}$ is defined (and thus we do not have to evaluate "$0 \cdot \infty$").

Another important properties of means is convexity resp. concavity. Power means $\mathfrak{M}_\alpha$ with $\alpha > 1$ are strictly convex, while power means $\mathfrak{M}_\alpha$ with $\alpha < 1$ are strictly concave; the borderline case $\mathfrak{M}_1 = \mathfrak{A}$ is linear (convex and concave).

Lemma 7.3.3. *For every concave mean M we have $M \leq \mathfrak{A}$.*

Proof. By Jensen's inequality (see [HUL04]) we have $M(a) = \sum_{\pi \in S_k} \frac{1}{k!} M(\pi * a) \leq M(\sum_{\pi \in S_k} \frac{1}{k!} \cdot (\pi * a)) = M(\mathfrak{A}(a) \cdot \vec{1}) = \mathfrak{A}(a)$. □

We now consider the means associated with the τ-function. In the following $\log = \log_e$ denotes the natural logarithm (to base e).

Definition 7.3.5. For $k \geq 2$ the map $\mathfrak{T}_k : \mathcal{BT}^{(k)} \to \mathbb{R}_{>0}$ is defined by

$$\mathfrak{T}_k(t) := \frac{\log(k)}{\log(\tau(t))} = \log_{\tau(t)}(k).$$

Note that while a smaller $\tau(t)$ indicates a better t, for the τ-mean $\mathfrak{T}$ the larger $\mathfrak{T}_k(t)$ is the better t is, but this only holds for fixed k, and due to the normalisation the τ-means of tuples of different lengths cannot be compared.

Theorem 7.3.4. *Consider $k \geq 2$.*

1. *$\mathfrak{T}_k$ is a strictly concave homogeneous mean.*
2. *We have $\mathfrak{M}_{2-k}(t) \leq \mathfrak{T}_k(t) \leq \mathfrak{A}(t)$, with equalities iff t is constant. Especially we have $\mathfrak{G}(t) \leq \mathfrak{T}_2(t) \leq \mathfrak{A}(t)$.*
3. *For $k \geq 3$ it is $\mathfrak{T}_k$ 0-dominated and ∞-ignoring, while $\mathfrak{T}_2$ is 0-dominated as well as ∞-dominated, whence only defined for $t \in \overline{\mathbb{R}}_{\geq 0}^2 \setminus \{(0, \inf), (\inf, 0)\}$.*

In general we have that if a positive function $f : C \to \mathbb{R}_{>0}$ is "reciprocal-concave", that is, $1/f$ is concave, then f is log-convex (but not vice versa), that is, $\log \circ f$ is convex. Furthermore, if f is log-convex then f is convex (but not vice versa).

Corollary 7.3.5. *Consider $k \geq 2$.*

1. *τ_k is strictly reciprocal-log-concave for $k \geq 2$, that is, the map $t \in \mathcal{BT}^{(k)} \mapsto 1/\log(\tau(t)) \in \mathbb{R}_{>0}$ is strictly concave. Thus τ is strictly log-log-convex.*
2. *For $t \in \mathcal{BT}^{(k)}$ we have $\tau(\mathfrak{A}(t) \cdot \vec{1}) = k^{1/\mathfrak{A}(t)} \leq \tau(t) \leq k^{1/\mathfrak{M}_{2-k}(t)} = \tau(\mathfrak{M}_{2-k}(t) \cdot \vec{1})$.*

Remarks:

1. Part 2 says that replacing all entries in a branching tuple by the arithmetic mean of the branching tuple improves (decreases) the τ-value, while replacing all entries by the geometric mean ($k = 2$) resp. harmonic mean

($k = 3$) impairs (increases) the τ-value. The case $k = 2$ of this inequality was shown in [KL98] (Lemma 5.6). For the use of the lower and upper bounds in heuristics see Subsection 7.6.
2. Computation of $\tau(t)$ can only be accomplished numerically (except of a few special cases), and a suitable method is the Newton-method (for computing the root of $\chi(t)(x) - 1$), where using the lower bound $|t|^{1/\mathfrak{A}(t)}$ as initial value performs very well (guaranteeing monotonic convergence to $\tau(t)$).

7.3.4. Associating probability distributions with branching tuples

Definition 7.3.6. Given a branching tuple $a = (a_1, \dots, a_k)$, the branching tuple $\tau^{\mathrm{p}}(a) \in \mathcal{BT}^{(k)}$ is defined by $\tau^{\mathrm{p}}(a)_i := \tau(a)^{-a_i}$ for $i \in \{1, \dots, k\}$.

Remarks:

1. By definition we have $\Sigma(\tau^{\mathrm{p}}(a)) = 1$, and thus $\tau^{\mathrm{p}}(a)$ represents a probability distribution (on $\{1, \dots, k\}$).
2. For $\lambda \in \mathbb{R}_{>0}$ we have $\tau^{\mathrm{p}}(\lambda \cdot a) = \tau^{\mathrm{p}}(a)$, and thus $\tau^{\mathrm{p}}(a)$ only depends on the relative sizes of the entries a_i, and not on their absolute sizes.
3. For fixed k we have the map $\tau_k^p : \mathcal{BT}^{(k)} \to \mathrm{int}(\sigma_{k-1}) \subset \mathcal{BT}^{(k)}$ from the set of branching tuples $(a_1, \dots, a_k)$ of length k to the interior of the $(k-1)$-dimensional standard simplex σ_{k-1}, that is, to the set of all branching tuples $(p_1, \dots, p_k) \in \mathcal{BT}^{(k)}$ with $p_1 + \cdots + p_k = 1$. We have already seen that $\tau_k^p(\lambda \cdot a) = \tau_k^p(a)$ holds. Furthermore τ_k^p is surjective, i.e., every probability distribution of size k with only nonzero probabilities is obtained, with $(\tau_k^p)^{-1}((p_1, \dots, p_k)) = \mathbb{R}_{>0} \cdot (-\log(p_1), \dots, -\log(p_k))$.

7.4. Estimating tree sizes

Now we turn to the main application of branching tuples and the τ-function, the estimation of tree sizes. We consider rooted trees (T, r), where T is an acyclic connected (undirected) graph and r, the root, is a distinguished vertex of T. Since we are considering only rooted trees here, we speak in the sequel just of "trees" T with root $\mathrm{rt}(T)$ and vertex set $V(T)$. We use $\#\mathrm{nds}(T) := |V(T)|$ for the number of nodes of T, while $\#\mathrm{lvs}(T) := |\mathrm{lvs}(T)|$ denotes the number of leaves of T.

7.4.1. Notions for trees

For a node $v \in V(T)$ let $\mathrm{d}_T(v)$ be the *depth* of v (the length of the path from the root to v), and for $i \in \{0, \dots, \mathrm{d}(v)\}$ let $T(i, v)$ be the vertex with depth i on the path from the root to v (so that $T(0, v) = \mathrm{rt}(T)$ and $T(\mathrm{d}(v), v) = v$). For a node $v \in V(T)$ we denote by T_v the subtree of T with root v. Before turning to our main subject, measuring the size of trees, some basic strategic ideas should be pointed out:

- Trees are considered as "static", that is, as given, not as evolving; the main advantage of this position is that it enables us to consider the "real" backtracking trees, in contrast to the standard method of ignoring the real object and only to consider approximations given by recursion equations.

- The number of leaves is a measure easier to handle than the number of nodes: When combining trees under a new root, the number of leaves behaves additively, while the number of nodes is bigger by one node (the new root) than the sum. Reductions, which correspond to nodes with only a single successor, are being ignored in this way. For binary trees (every inner node as exactly two children) we have $\#\mathrm{nds}(T) = 2\,\#\mathrm{lvs}(T) - 1$. And finally, heuristics in a SAT solver aim at reducing the number of conflicts found, that is, the number of leaves.
- All leaves are treated equal (again, this corresponds to the point of view of the heuristics).

7.4.2. Adorning trees with probability distributions

Consider a finite probability space (Ω, P), i.e., a finite set Ω of "outcomes" together with a probability assignment $P : \Omega \to [0,1]$ such that $\sum_{\omega \in \Omega} P(\omega) = 1$; we assume furthermore that no element has zero probability ($\forall\, \omega \in \Omega : P(\omega) > 0$). The random variable P^{-1} on the probability space (Ω, P) assigns to every outcome ω the value $P(\omega)^{-1}$, and a trivial calculation shows that the expected value of P is the number of outcomes:

$$\mathrm{E}(P^{-1}) = \sum_{\omega \in \Omega} P(\omega) P(\omega)^{-1} = |\Omega|. \tag{7.1}$$

So the random variable P^{-1} associates to every outcome ω a guess $P^{-1}(\omega)$ on the (total) number of outcomes, and the expected value of these guesses is the true total number of all outcomes. Thus, via sampling of P^{-1} we obtain an estimation on $|\Omega|$. In the general context this seems absurd, since the probabilities of outcomes are normally not given a priori, however in our application, where the outcomes of the probability experiment are the leaves of the search tree, we have natural ways at hand to calculate for each outcome its probability. We remark that for $r \in \mathbb{R}_{\geq 1}$ from (7.1) we get for the r-th moment the lower bound $\mathrm{E}((P^{-1})^r) = \mathrm{E}((P^{-r})) \geq |\Omega|^r$ (by Jensen's inequality).

Definition 7.4.1. For trees T we consider *tree probability distributions* $\mathfrak{P}$, which assign to every edge (v, w) in T a probability $\mathfrak{P}((v,w)) \in [0,1]$ such that for all inner nodes v we have $\sum_{w \in \mathrm{ds}_T(v)} \mathfrak{P}((v,w)) = 1$, that is, the sum of outgoing probabilities is 1; we assume furthermore, that no edge gets a zero probability. We obtain the associated probability space (Ω_T, P), where $\Omega_T := \mathrm{lvs}(T)$, that is, the outcomes are the leaves of T, which have probability

$$\mathrm{P}(v) := \prod_{i=0}^{\mathrm{d}(v)-1} \mathfrak{P}((T(i,v), T(i+1,v))). \tag{7.2}$$

The edge-probabilities being non-zero just means that no outcome in this probability space has zero probability (which would mean it would be ignored).

Equation (7.2) makes sense for any vertex $v \in V(T)$, and $\mathrm{P}(v)$ is then to be interpreted as the event that an outcome is a leaf in the subtree of T with root v (that is, $\mathrm{P}(v) = \sum_{w \in \mathrm{lvs}(T_v)} \mathrm{P}_T(w)$); however we emphasise that the values $\mathrm{P}(v)$ for inner nodes v are only auxiliary values. From (7.1) we obtain:

Lemma 7.4.1. *For every finite rooted tree T and every tree probability distribution $\mathfrak{P}$ for T we have for the associated probability space Ω_T and the random variable $\mathrm{P}^{-1} : \Omega_T \to]0,1]$:*

$$\min \mathrm{P}^{-1} = \min_{v \in \mathrm{lvs}(T)} \mathrm{P}(v)^{-1} \leq \#\mathrm{lvs}(T) = \mathrm{E}(\mathrm{P}^{-1}) \leq \max_{v \in \mathrm{lvs}(T)} \mathrm{P}(v)^{-1} = \max \mathrm{P}^{-1} .$$

Corollary 7.4.2. *Under the assumptions of Lemma 7.4.1 the following assertions are equivalent:*

1. $\min \mathrm{P}^{-1} = \#\mathrm{lvs}(T)$.
2. $\#\mathrm{lvs}(T) = \max \mathrm{P}^{-1}$.
3. P *is a uniform distribution (all leaves have the same probability).*

Lemma 7.4.1 opens up the following possibilities for estimating the size of a tree T, given a tree probability distribution $\mathfrak{P}$:

1. Upper bounding $\max \mathrm{P}^{-1}$ we obtain an upper bound on $\#\mathrm{lvs}(T)$, while lower bounding $\min \mathrm{P}^{-1}$ we obtain a lower bound on $\#\mathrm{lvs}(T)$.
2. Estimating $\mathrm{E}(\mathrm{P}^{-1})$ by sampling we obtain an estimation of $\#\mathrm{lvs}(T)$.

By Corollary 7.4.2, in each case a tree probability distribution $\mathfrak{P}$ yielding a uniform distribution p on the leaves is the most desirable distribution (the lower and upper bounds coincide with the true value, and only one path needs to be sampled). It is easy to see that each tree has exactly one such "optimal tree probability distribution":

Lemma 7.4.3. *Every finite rooted tree T has exactly one tree probability distribution $\mathfrak{P}$ which induces a uniform probability distribution* P *on the leaves, and this* ***canonical tree probability distribution*** $\mathfrak{CP}_T$ *is given by*

$$\mathfrak{CP}_T((v,w)) = \frac{\#\mathrm{lvs}(T_w)}{\#\mathrm{lvs}(T_v)}$$

for $v \in V(T)$ and $w \in \mathrm{ds}_T(v)$.

The canonical tree probability distribution $\mathfrak{CP}_{T_v}$ on a subtree T_v of T (for $v \in V(T)$) is simply obtained by restricting $\mathfrak{CP}_T$ to the edges of T_v (without change).

7.4.3. The variance of the estimation of the number of leaves

If the leaf probabilities vary strongly, then the variance of the random variable P^{-1} will be very high, and a large number of samples is needed to obtain a reasonable estimate on the number of leaves. So we should consider more closely the variance

$\mathrm{Var}(\mathrm{P}^{-1}) = \mathrm{E}((\mathrm{P}^{-1} - \#\mathrm{lvs}(T))^2) = \mathrm{E}(\mathrm{P}^{-2}) - \#\mathrm{lvs}(T)^2 \in \mathbb{R}_{\geq 0}$ of the random variable P^{-1}. By definition, the variance is 0 if and only if the probability distribution is uniform, that is, iff $\mathfrak{P}$ is the canonical tree probability distribution on T. To estimate $\mathrm{Var}(\mathrm{P}^{-1})$ one needs to estimate $\mathrm{E}(\mathrm{P}^{-2})$, that is, the second moment of P^{-1}. By definition we have $\mathrm{E}(\mathrm{P}^{-2}) = \sum_{v \in \mathrm{lvs}(T)} \mathrm{P}(v) \cdot \mathrm{P}(v)^{-2} = \sum_{v \in \mathrm{lvs}(T)} \mathrm{P}(v)^{-1}$. So $\mathrm{E}(\mathrm{P}^{-2})$ is just the sum over all estimations on $\#\mathrm{lvs}(T)$ we obtain from the probability distribution P. Somewhat more efficiently, we can calculate all moments of P^{-1} recursively (using a number of arithmetical operations which is linear in $\#\mathrm{nds}(T)$) as follows, where we use $\mathfrak{P}_{T_v}$ for the restriction of the tree probability distribution $\mathfrak{P} = \mathfrak{P}_T$ to subtree T_v (unchanged), while P_{T_v} is the probability distribution induced by $\mathfrak{P}_{T_v}$ (which is not the restriction of P_T; for $w \in \mathrm{lvs}(T_v)$ we have $\mathrm{P}_{T_v}(w) = \mathrm{P}_T(v)^{-1} \cdot \mathrm{P}_T(w)$). Trivial calculations show:

Lemma 7.4.4. *For a finite rooted tree T, a tree probability distribution $\mathfrak{P}$ on T and $r \in \mathbb{R}_{\geq 0}$ we can recursively compute the r-th moment $\mathrm{E}(\mathrm{P}^{-r})$ of P^{-1} by*

- *If T is trivial (i.e., $\#\mathrm{nds}(T) = 1$), then we have $\mathrm{E}(\mathrm{P}_T^{-r}) = 1$.*
- *Otherwise $\mathrm{E}(\mathrm{P}_T^{-r}) = \sum_{v \in \mathrm{ds}(\mathrm{rt}(T))} \mathfrak{P}_T((\mathrm{rt}(T), v))^{1-r} \cdot \mathrm{E}(\mathrm{P}_{T_v}^{-r})$.*

Following [Knu75] (Theorem 3), we give an estimation on the variance of P^{-1} if the derived transition probabilities on the edges are within a factor α of the canonical tree probability distribution (recall Lemma 7.4.3).

Lemma 7.4.5. *For a finite rooted tree T and a tree probability distribution $\mathfrak{P}$ on T, which fulfils $\mathfrak{P}_T^{-1} \leq \alpha \cdot \mathfrak{C}\mathfrak{P}_T^{-1}$ for some $\alpha \in \mathbb{R}_{\geq 1}$, we have $\mathrm{E}(\mathrm{P}_T^{-r}) \leq \alpha^{(r-1)\cdot \mathrm{ht}(T)} \cdot \#\mathrm{lvs}(T)^r$ for all $r \in \mathbb{R}_{\geq 1}$.*

Corollary 7.4.6. *Under the same assumptions as in Lemma 7.4.5 we have*

$$\mathrm{Var}(\mathrm{P}_T^{-1}) \leq (\alpha^{\mathrm{ht}(T)} - 1) \cdot \#\mathrm{lvs}(T)^2.$$

All the considerations of this section can be generalised to the case, where we also take inner nodes of the tree into account, and where we have an arbitrary cost function which assigns to every node of the tree its cost (in this section we considered the cost function which assigns every leaf the cost 1, while every inner node gets assigned cost 0). See [Kul08a] for details.

7.4.4. The tau-method

In the previous subsection we developed a method of estimating tree sizes, assuming a given tree probability distribution which assigns to every edge a transition probability. Now in this subsection we discuss how to obtain such transition probabilities via the help of "distances" and "measures". The basic idea is to attach a distance $d((u, v))$ to the edge (u, v), measuring how much "progress" was achieved via the transition from u to v, and where a standard method for obtaining such distances is to use a measure μ of "complexity" via $d((u, v)) := \mu(u) - \mu(v)$.

Definition 7.4.2. A *distance* d on a finite rooted tree T is a map $d : E(T) \to \mathbb{R}_{>0}$ which assigns to every edge (v, w) in T a positive real number, while a *measure* is a map $\mu : V(T) \to \mathbb{R}$ such that $\Delta\mu$ is a distance, where $\Delta\mu((v, w)) := \mu(v) - \mu(w)$. For a distance d we define the measures $\min \Sigma d$ and $\max \Sigma d$ on T, which assign to every vertex $v \in V(T)$ the minimal resp. maximal sum of d-distances over all paths from v to some leave.

The τ-function yields a canonical method to associate a tree probability distribution to a distance on a tree as follows.

Definition 7.4.3. Consider a rooted tree T together with a distance d. For an inner node v of T we obtain an associated branching tuple $d(v)$ modulo permutation; assuming that T is ordered, i.e., we have a sorting $\mathrm{ds}_T(v) = \{v_1, \dots, v_k\}$, we obtain a concrete branching tuple $d(v) := (d(v, v_1), \dots, d(v, v_k))$. The associated tree probability distribution $\mathfrak{P}_d$ is given by

$$\mathfrak{P}_d((v, v_i)) := \tau^{\mathrm{P}}(d(v))_i$$

(recall Definition 7.3.6).

By definition we have for $\lambda \in \mathbb{R}_{>0}$ that $\mathfrak{P}_{\lambda \cdot d} = \mathfrak{P}_d$. By Remark 3 to Definition 7.3.6 for every tree probability distribution $\mathfrak{P}$ for T there exist distances d on T with $\mathfrak{P} = \mathfrak{P}_d$, and d is unique up to scaling of the branching tuples at each inner node (but each inner node can be scaled differently).

Given a distance d on a tree T, in order to apply Lemma 7.4.1 we need to estimate $\min \mathrm{P}_d^{-1}$ and $\max \mathrm{P}_d^{-1}$, where P_d is the probability distribution induced by $\mathfrak{P}_d$ on the leaves of T according to Definition 7.4.1.

Definition 7.4.4. For a rooted tree (T, r) with a distance d let $\min \tau(d) := +\infty$ and $\max \tau(d) := 1$ in case T is trivial (consists just of r), while otherwise $\min \tau(d) := \min_{v \in V(T) \setminus \mathrm{lvs}(T)} \tau(d(v))$ and $\max \tau(d) := \max_{v \in V(T) \setminus \mathrm{lvs}(T)} \tau(d(v))$.

Lemma 7.4.7. *Consider a rooted tree (T, r) together with a distance d. For the induced probability distribution P_d on the leaves of T we have:*

1. $(\min \tau(d))^{\min \Sigma d(r)} \leq \min \mathrm{P}_d^{-1}$.
2. $\max \mathrm{P}_d^{-1} \leq (\max \tau(d))^{\max \Sigma d(r)}$.

Proof. We prove Part 1 (the proof for Part 2 is analogous). If T is trivial then the assertion is trivial, so assume that T is non-trivial. Let $\tau_0 := \min \tau(d)$.

$$\min \mathrm{P}_d^{-1} = \min_{v \in \mathrm{lvs}(T)} \mathrm{P}_d(v)^{-1} = \min_{v \in \mathrm{lvs}(T)} \prod_{i=0}^{d(v)-1} \mathfrak{P}_d(T(i, v), T(i+1, v))^{-1} =$$

$$\min_{v \in \mathrm{lvs}(T)} \prod_{i=0}^{d(v)-1} \tau(d(T(i, v)))^{d(T(i,v), T(i+1,v))} \leq$$

$$\min_{v \in \mathrm{lvs}(T)} \prod_{i=0}^{d(v)-1} \tau_0^{d(T(i,v), T(i+1,v))} = \min_{v \in \mathrm{lvs}(T)} \tau_0^{\sum_{i=0}^{d(v)-1} d(T(i,v), T(i+1,v))} = \tau_0^{\min \Sigma d(r)}.$$

□

Lemma 7.4.1 together with Lemma 7.4.7 yields immediately the following fundamental method for estimating tree sizes.

Theorem 7.4.8. *Consider a rooted tree (T, r). For a distance d on (T, r) we have*

$$(\min \tau(d))^{\min \Sigma d(r)} \leq \#\mathrm{lvs}(T) \leq (\max \tau(d))^{\max \Sigma d(r)}.$$

And for a measure μ on (T, r) which is 0 *on the leaves we have*

$$(\min \tau(\Delta\mu))^{\mu(r)} \leq \#\mathrm{lvs}(T) \leq (\max \tau(\Delta\mu))^{\mu(r)}.$$

Remarks:

1. So upper bounds on tree sizes are obtained by Theorem 7.4.8 through
   - upper bounds on the τ-values on the branching tuples at each inner node of the tree
   - and upper bounds on the maximal sum of distances amongst paths in the tree,

   where the latter can be obtained via the root-measure in case the distances are measure-differences.
2. The general method of Theorem 7.4.8 was introduced by [Kul99b] (Lemma 8.2 there; see Section 7.7.3 here for more on the topic of theoretical upper bounds), while in [KL97, KL98] one finds direct (inductive) proofs.

The ideal measure on a tree makes the bounds from Theorem 7.4.8 becoming equal:

Lemma 7.4.9. *Consider a non-trivial rooted tree (T, r). Then the following assertions are equivalent for a measure μ on (T, r) which is* 0 *on the leaves:*

1. *There exists $\lambda \in \mathbb{R}_{>0}$ with $\forall\, v \in V(T) : \mu = \lambda \cdot \log(\#\mathrm{lvs}(T_v))$.*
2. $(\min \tau(\Delta\mu))^{\mu(r)} = \#\mathrm{lvs}(T)$.
3. $(\max \tau(\Delta\mu))^{\mu(r)} = \#\mathrm{lvs}(T)$.
4. $\min \tau(\Delta\mu) = \max \tau(\Delta\mu)$.

If one of these (equivalent) conditions are fulfilled then $\mathfrak{P}_{\Delta\mu} = \mathfrak{C}\mathfrak{P}_T$ (the canonical tree probability distribution; recall Lemma 7.4.3).

So a good distance d on a rooted tree T has to achieve two goals:

1. Locally, for each inner node v with direct successors $\mathrm{ds}(v) = \{v_1, \ldots, v_k\}$ the relative proportions of the distances $d(v, v_i)$ must mirror the sizes of the subtrees hanging at v_i (the larger the subtree the smaller the distance).
2. Globally, the scaling of the different τ-values at inner nodes must be similar.

As we have seen in Lemma 7.4.9, if the second condition is fulfilled perfectly, then also the first condition is fulfilled (perfectly), while the other direction does not hold (the first condition can be achieved with arbitrary scaling of single branching tuples). The problem of constructing good distance functions (from a general point of view) is further discussed in Section 7.8.

After having seen that the τ-function can be put to good use, in the following section we show that at least the linear quasi-order induced on branching tuples by the τ-function (the smaller the τ-value the "better" the tuple) follows from very fundamental requirements on the evaluation of branchings.

7.5. Axiomatising the canonical order on branching tuples

On $\mathcal{BT}$ we have a natural linear quasi-order given by $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$ for $a, b \in \mathcal{BT}$. Here we show how this order follows from simple intuitive axioms (extending [Kul98]).

Definition 7.5.1. A relation $\leq$ on $\mathcal{BT}$ is called a **canonical branching order** if it fulfils the six properties (TQO), (CMP), (P), (W), (M) and (Con), which are the following conditions (for all branching tuples a, b, c and all permutations π), where the induced equivalence relation $a \sim b :\Leftrightarrow a \leq b \wedge b \leq a$ and the induced strict order $a < b :\Leftrightarrow a \leq b \wedge a \not\sim b$ are used. First the five elementary conditions are stated:

(TQO) ("Total Quasi-order") $\leq$ is a total quasi-order on $\mathcal{BT}$, that is, $a \leq a$, $a \leq b \wedge b \leq c \Rightarrow a \leq c$ and $a \leq b \vee b \leq a$ for all branching tuples a, b, c.
(CMP) ("Composition") $a \leq b \Longrightarrow (a \leq a \mathbin{/\!\!\backslash\!\!\backslash} b \leq b) \ \wedge \ (a \leq b \mathbin{/\!\!\backslash\!\!\backslash} a \leq b)$.
(P) ("Permutation") $\pi * a \sim a$.
(W) ("Widening") $a \sqsubset b \Rightarrow a < b$.
(M) ("Monotonicity") If $k := |a| = |b| \geq 2$, $\forall\, i \in \{1, \dots, k\} : a_i \leq b_i$ and $\exists\, i \in \{1, \dots, k\} : a_i > b_i$, then $a < b$.

Now for $a \in \mathcal{BT}$ let $\gamma_a : \mathbb{R}_{>0} \to \mathcal{BT}$ be defined by $\gamma_a(x) := a\,;(x)$ ("left translation"). So (M) just expresses that for $a \in \mathcal{BT}$ the map γ_a is strictly decreasing. And extend $\gamma_a : \mathbb{R}_{>0} \to \mathcal{BT}$ to $\gamma_a : \overline{\mathbb{R}}_{>0} \to \mathcal{BT}$ by $\gamma(+\infty) := a$. The remaining condition is:

(Con) ("Continuity") For $a \in \mathcal{BT}$ the map $\gamma_a : \overline{\mathbb{R}}_{>0} \to \mathcal{BT}$ is continuous with regard to the natural topology on $\overline{\mathbb{R}}_{>0}$ and the order topology on $\mathcal{BT}$, i.e.:

(Con1) For $x \in \mathbb{R}_{>0}$ and $b, c \in \mathcal{BT}$ with $b < \gamma_a(x) < c$ there is $\delta \in \mathbb{R}_{>0}$ such that for $x' \in \mathbb{R}_{>0}$ with $|x - x'| < \delta$ we have $b < \gamma_a(x') < c$.
(Con2) For $b \in \mathcal{BT}$ with $b > a$ there is $x_0 \in \mathbb{R}_{>0}$ such that for $x' \in \mathbb{R}_{>x_0}$ we have $b > \gamma_a(x')$.

Remarks:

1. The intuitive meaning of "$a \leq b$" is that "in general", that is, "if nothing else is known", branching tuple a is at least as good as b (doesn't lead to larger branching trees). The main result of this section is that actually there is exactly one canonical branching order.

2. (TQO) expresses the comparability of branching tuples; the order does not fulfil antisymmetry (i.e., $a \leq b \wedge b \leq a \Rightarrow a = b$), since for example, as stated in (P), permutation doesn't change the value of a branching, and via (CMP) also composition of a branching tuple with itself doesn't change its value.
3. (CMP) states that if a is at least as good as b, then $a \mathbin{/\!\!\backslash\!\!\backslash} b$ as well as $b \mathbin{/\!\!\backslash\!\!\backslash} a$ are “compromises”, improving b and impairing a.
4. (P) says permutation does not change anything essential.
5. (W) requires that adding branches to a branching impairs the branching.
6. (M) states that increasing some component of a branching tuple of width at least two strictly improves the branching tuple.
7. Finally (Con) states that sufficiently small changes in one component yield only small changes in the “value” of the tuple.

Lemma 7.5.1. *The linear quasi-order on $\mathcal{BT}$ given by $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$ for $a, b \in \mathcal{BT}$ is a canonical branching order.*

Lemma 7.5.2. *Consider a canonical branching order $\leq$ on $\mathcal{BT}$. Then for $a, b \in \mathcal{BT}$ with $\tau(a) < \tau(b)$ we have $a < b$.*

Theorem 7.5.3. *There is exactly one canonical branching order on branching tuples, given by $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$ for all $a, b \in \mathcal{BT}$.*

Proof. Consider a canonical branching order $\leq$. We have to show that for all $a, b \in \mathcal{BT}$ we have $a \leq b \Leftrightarrow \tau(a) \leq \tau(b)$. By Lemma 7.5.1 we know the direction from right to left. So assume that $\leq$ is a canonical branching order, and consider $a, b \in \mathcal{BT}$. If $a \leq b$ holds, then $\tau(a) > \tau(b)$ by Lemma 7.5.2 would imply $a > b$ contradicting the assumption. So assume now (finally) $\tau(a) \leq \tau(b)$, and we have to show that $a \leq b$ holds. If $a > b$ would be the case, then by (Con1) there exists $\varepsilon \in \mathbb{R}_{>0}$ with $a \mathbin{/\!\!\backslash\!\!\backslash} (\varepsilon) > b$, however we have $\tau(a \mathbin{/\!\!\backslash\!\!\backslash} (\varepsilon)) < \tau(a) \leq \tau(b)$, and thus by Lemma 7.5.2 it would hold $a \mathbin{/\!\!\backslash\!\!\backslash} (\varepsilon) < b$. □

For branching tuples with rational entries the canonical branching order can be decided in polynomial time (in the binary representation) by using the decidability of the first-order theory of the real numbers ([Ren92]). The approach of this section can be generalised by considering only tuples of length at most k resp. of length equal k, and formulating the axioms in such a way that all occurring branching tuples are in the restricted set. In the light of the questions arising in the subsequent Section 7.6 about projections for branching tuples of restricted width such generalisations seem to be worth to study.

7.6. Alternative projections for restricted branching width

The argumentation of Subsection 7.5 depends on considering branching tuples of arbitrary length. The strength of the τ-function is that it imposes a consistent scaling for tuples of different sizes, and Theorem 7.5.3 shows that the order induced by the τ-function (the canonical order) is the only reasonable order if branching tuples of arbitrary width are considered. Now if we consider only branching tuples of some constant length, then other choices are possible. Practical experience has shown that for a binary branching tuple (a_1, a_2) maximising the product $a_1 \cdot a_2$ yields good results, and this projection is universally used now (ignoring tie-breaking aspects here), while maximising the sum $a_1 + a_2$ has been shown by all experiments to perform badly. We are now in a position to give theoretically founded explanations:

1. The general rule is that $\tau(a_1, a_2)$ should be minimised.
2. If computation of $\tau(a_1, a_2)$ is considered to be too expensive, then the approximations from Corollary 1, Part 2 could be used, which amount here to either maximise the arithmetic mean $\mathfrak{A}(a_1, a_2)$ or to maximise the geometric mean $\mathfrak{G}(a_1, a_2)$. Now maximising $\mathfrak{A}(a_1, a_2)$ is equivalent to maximise the sum $a_1 + a_2$, while maximising the geometric mean $\mathfrak{G}(a_1, a_2)$ is equivalent to maximising the product $a_1 \cdot a_2$.
3. So maximising the sum $a_1 + a_2$ means to minimise a lower bound on the τ-value, while maximising the product $a_1 \cdot a_2$ means minimising an upper bound on the τ-value — it appears now that the second choice is more meaningful, since it amounts to minimise an upper bound on the tree size, while minimising a lower bound on the tree size leads to nowhere.

A more quantitative explanation is given by the following lemma, which shows that the product yields a better approximation to the τ-mean than the sum.

Lemma 7.6.1. *We have $\mathfrak{A}(a_1, a_2) - \mathfrak{T}(a_1, a_2) \geq \mathfrak{T}(a_1, a_2) - \mathfrak{G}(a_1, a_2)$ for all $a_1, a_2 \in \mathbb{R}_{>0}$, with equality iff $a_1 = a_2$ (and in this case both sides are zero).*

Especially for branching tuples with more than two entries the τ-function appears as the canonical choice; if approximations are sought then the product (corresponding to the geometrical mean) can no longer be used, but the general upper bound from Corollary 1, Part 2 is a candidate.[3] It is not clear whether for branching tuples of constant width the canonical order is always the superior choice (and using for example the product-rule for binary branches is just a reasonable approximation), or whether there might be special (but still "general") circumstances under which other orders are preferable. Considering again binary branchings, for the canonical order the branching tuples $(1, 5)$ and $(2, 3)$ are equivalent, but they are distinguished by the product rule which favours $(2, 3)$. This is in fact a general property of the product rule (which, as already said, is only sensible for binary tuples) that it favours more symmetric tuples (compared to the canonical order).

[3]The unsuitability of the product for branching widths at least 3 can be seen for example by the fact that the product is ∞-dominated (i.e., if one component goes to infinity, so does the product), while the τ-function is ∞-ignorant — if one branch is very good, then this does not mean that the whole branching is very good, but only that this branch doesn't contribute to the overall cost.

7.7. How to select distances and measures

After having laid the foundations, in this section we consider now the basic "branching schemes" used by such (practical) SAT solvers employing branching algorithms. The essential ingredient of theses "schemes" (made more precise in Subsection 7.7.2) is the underlying distance function, however obviously the whole of the algorithm is important, and the aspects related to the branching heuristics are condensed in these "schemes".[4] The main purpose is to give an overview on the relevant measures $\mu(F)$ of "absolute problem complexity" or of distances $d(F, F')$ as measurements of "relative problem complexity", as core ingredients of the heuristical melange found in a SAT solver. This is intertwined with especially the reduction process, and we discuss this aspect further in Subsection 7.7.2. A potential source for future heuristics is the literature on worst-case upper bounds, and we make some comments in Subsection 7.7.3. Then in Subsection 7.7.4 we give a "rational reconstruction" of the branching schemes currently used in practice, which actually all can be explained as "maximising the number of new clauses" — this branching rule in its pure form was implemented by the `OKsolver`-2002 (see [Kul02]), and is presented in Subsection 7.7.4.1, while the historical development in practical SAT solving leading to this rule is discussed in Subsection 7.7.4.2. For background on "look-ahead SAT solvers" in general see Chapter 5 of this handbook.

7.7.1. Some basic notations for clause-sets

Applying the abstract ideas and notions to concrete problems, now we need to consider more closely actual problem instances. In the sequel we use the following notations for (multi-)clause-sets F:

- $n(F) := |\mathrm{var}(F)|$ is the number of variables (actually occurring in F).
- $c(F) := |F|$ is the number of clauses, and more specifically $c_k(F)$ for $k \in \mathbb{N}_0$ is the number of clauses of length (exactly) k.
- $\ell(F) = \sum_{C \in F} |C| = \sum_{k \in \mathbb{N}_0} c_k(F) \cdot k$ is the number of literal occurrences.
- The maximal clause-length is denoted by $\mathrm{rank}(F)$.
- For a partial assignment φ the result of applying φ to F is denoted by $\varphi * F$ (eliminating satisfied clauses, and removing falsified literals from the remaining clauses).
- The partial assignment (just) assigning value ε to variable v is denoted by $\langle v \to \varepsilon \rangle$.
- $r_k(F)$ denotes generalised unit-clause propagation (see [Kul99a]), i.e., r_0 just reduces F to $\{\bot\}$ iff $\bot \in F$, r_1 is unit-clause propagation, r_2 is failed-literal propagation etc.

[4] As remarked earlier, yet heuristics for conflict-driven solvers are mainly "purely heuristical", and no theoretical foundation exists, whence we do not consider them here. Some general remarks on heuristics in this context are in [Kul08b]. From a theoretical angle, in [BKS04] guidance by structure has been considered (on an example), while [BKD$^+$04] considered the relation between tree decompositions and learning.

7.7.2. On the notion of "look-ahead"

An important concept in this context is the notion of "look-ahead", which is used in the literature in different ways, and so needs some discussion. A *reduction* is typically a function $r(F)$ for problem instances F, computable in polynomial time such that problem instance $r(F)$ is satisfiability equivalent to F. Now for example in [Fre95] "look-ahead" is used mainly instead of "reduction" (actually framed in the language of constraint satisfaction, as "constraint propagator"), and this reduction process can actually be integrated into the heuristics for choosing the branching variable. This integration of reduction and heuristics seems to be rather popular amongst current look-ahead architectures, however in this article, with its foundational emphasis, we need clean and precise distinctions, and so we use the following architecture of a recursive backtracking SAT solver:

1. The input is a problem instance F.
2. Via the *reduction* r the instance is reduced to $F' := r(F)$, including at least unit-clause propagation (iterated elimination of (all) unit-clauses).
3. If now F' is trivially satisfiable or unsatisfiable, as established by the *immediate-decision oracle*, then the result is returned (more complex tests for satisfiability or unsatisfiability can be integrated into the reduction).
4. The purpose of the *branching scheme* (for standard SAT solvers) is now to select a *branching variable* $v \in \mathrm{var}(F')$ such that branching using the two branches

$$\begin{tikzcd}
 & F' \arrow[dl] \arrow[dr] & \\
r(\langle v \to 0\rangle * F') & & r(\langle v \to 1\rangle * F')
\end{tikzcd}$$

   yields the "fastest" decision (applying the algorithm recursively to the branches). Note that from this point of view branching considers only reduced instances. For sequential execution the ordering of the branches is important, and is accomplished by the *branching ordering heuristics*. The tasks of the branching scheme and the branching ordering heuristics are accomplished as follows:

   (a) For each variable $v \in \mathrm{var}(F')$ and value $\varepsilon \in \{0,1\}$ the *look-ahead reduction* r' computes an approximation of $r(\langle v \to \varepsilon\rangle * F')$ as $F_v^\varepsilon := r'(\langle v \to \varepsilon\rangle * F')$, where r' is a weaker form of r.
   (b) Via the *distance function* d for each variable v the branching tuple $t_v := (d(F', F_v^0), d(F', F_v^1)) \in \mathbb{R}_{>0}^2$ is computed.
   (c) Via the *projection* ρ each branching tuple t_v is projected to a single real number $\rho(t_v)$, and the branching scheme selects a variable v_0 with minimal $\rho(t_{v_0})$ (possibly using additional distance functions to break ties).
   (d) The branching order finally is established by a *satisfiability estimator* $P(F_{v_0}^\varepsilon)$, a function which computes for each branch $F_{v_0}^\varepsilon$ an "approximation" of the chance of being satisfiable, and the branch with higher P-value is chosen first.

In order to separate reduction and branching scheme we require here that no reduction takes place in Step 4 (where the branching scheme performs its work), that is, no F_v^ε is trivially decidable (according to Step 3), and thus r must actually be stronger than r'. Often r is the "look-ahead version" of r', that is, if r' is the identity, then r is just unit-clause propagation, and if r' is unit-clause propagation then r is failed-literal elimination; in general, if $r' = r_k$ then often $r = r_{k+1}$. In practice typically the separation between reduction and heuristic is blurred, and the choice of the "branching heuristics" involves a mixture of choices for reduction, look-ahead reduction, distances, projections and satisfiability estimator, typically applied in an entangled way to improve efficiency, but for the correct understanding the above distinctions seem essential.

7.7.3. Branching schemes in theoretical algorithms

[Kul92, Kul99b] introduced the now dominant technique for proving worst-case upper bounds on NP-hard problems, called "measure-and-conquer" in [FGK05], which is based on Theorem 7.4.8 and the use of an appropriate "interesting" distance d or measure μ. For a recent work on upper bounds and measures see [Wah07], and Chapter 12 of this handbook for worst-case upper bounds in general, while in this chapter we only consider the particular strand of work in this area which influenced heuristics for SAT solving.

In [KL97, KL98] the basic measures n, c, ℓ for SAT decision have been considered for the first time systematically. [Kul99b], motivated by [Sch92], based his analysis on the (measure-based) distance $d^2 = \Delta m_k = \Delta n - \alpha \Delta z_k$, where z_k is a capped version of c_2 (in order to make c_2 and n comparable), and $\alpha \in \mathbb{R}_{>0}$ is to be optimised. Note that a *decrease* in n is favourable, while an *increase* in the number of binary clauses is favourable. An important refinement then replaces the distance d^2 by the distance d^3, which takes (depending on the parameter k) a fixed amount of *new* binary clauses into account (as improvement), while "ignoring" the number of eliminated binary clauses. Starting from this measure, by extensive experimentation the heuristic for the `OKsolver`-2002 was empirically "derived" ([Kul98]), arriving for 3-CNF at a distance function without magic numbers, which is just the number of *new binary clauses*.

That the distance Δn vanished finally in this development can be motivated as follows: Δn was used in [Kul99b] since an upper bound in the measure n was to be derived. But "heuristical" versions, which just care about making the tree smaller, without caring about accompanying (interpretable) upper bounds, can remove the factor n which imposed, due to the positivity requirement $d^3 > 0$, a cap on the number k of binary clauses to be taken into account — now arbitrary amounts of new binary clauses count! This branching scheme, generalised to arbitrary CNF will be further discussed in Subsection 7.7.4.1. It seems to be at this time the strongest "pure" heuristics known (for look-ahead solver, and used as basis for further heuristical extensions). And, by "magic coincidence", its basic content is closely related to the emerging heuristics from practical SAT solving, as explained in Subsection 7.7.4.2.

7.7.4. Branching schemes in practical algorithms

Now we turn to branching schemes as proposed (and used) in the realm of "practical SAT solving". First a few comments on very simple underlying distances, which might be useful for comparisons (or perhaps in the context of optimising combinations of distances, as considered in Section 7.8):

1. The trivial choice, an "arbitrary variable", is reflected by choosing a constant distance function. Here the "tie-braking rule" becomes decisive, and choosing a random variable, which for random instances is the same as choosing the first variable in a fixed (instance-independent) order, is considerably worse than choosing the first occurring variable in the clause-set, which for random instances in the average will choose the variable occurring most often.
2. Δn is perhaps the most basic non-trivial choice. This distance gets stronger with look-ahead, but for practical SAT solving it never played a role yet (different from the theoretical upper bounds as mentioned in Subsection 7.7.3). Δc and $\Delta \ell$ are more sensitive to the input[5], but still these distances appear as not suitable for practical SAT solving in general. Likely the basic problem with these measures is that they do not provide any guidance towards increased efficiency of the reduction process (unit-clause propagation and beyond).

We present the branching scheme of the `OKsolver`-2002 in Subsection 7.7.4.1 as the core rule: It represents a certain convergence of theoretical considerations coming from worst-case upper bounds (see Subsection 7.7.3) with practical developments (see Subsection 7.7.4.2), it has been deliberately kept free from purely heuristical considerations (as much as possible — only the clause-weights appear to be unavoidably of heuristical nature), and this one rule actually represents the core of modern branching rules for look-ahead solvers. As explained in Subsection 7.7.3, the underlying distance started from a combination of Δn and the number of new 2-clauses (for inputs in 3-CNF), but experimental results forced Δn to leave, and the result is close to the rule `dsj` as a result of the development starting with the Jeroslow-Wang rule — however the novelty is, and this came from the theoretical investigations in worst-case analysis, to view the whole process as applying a distance function, with the aim of maximising the number of *new clauses*.[6] In the literature on practical SAT solving, this target (creating as many strong constraints as possible) has been first discussed in [Li99].

There have been a few attempts of going beyond the underlying distance:

1. In [Ouy99], Chapter 3, we find a scheme to dynamically adjust the weights of clauses of length 2, leading to branching rule "B". However, since this considers the "old" clauses, it really means new clauses of length 1, i.e.,

[5]where for $c \geq n$ to hold, reduction by matching autarkies is needed (see [KL98, Kul03]

[6]The explanations in [HV95] for the first time concentrated on the role of unit-clause propagations, however from the point of view of simplifying the current formula, not, as the distance functions emphasises, from the point of view of the emergence of *future reductions*.

unit-clauses, which from our point of view shouldn't be considered in this form by the branching rule, but it should be considered by the (r_1) look-ahead. So this scheme seems only to be another approximation of the look-ahead, where at least for "look-ahead solvers" the r_1-look-ahead seems mandatory now.

2. Van Maaren et al considered non-linear approximations of CNF in order to gain a deeper, geometrical understanding of the problem instance. These efforts culminated in the branchings rule discussed in [vW00], where branching rule "MAR" is presented. The main idea is to derive for the residual clause-set in a natural way an n-dimensional ellipsoid (where n is the number of variables of the current clause-set), where each variable corresponds to one axis of the ellipsoid, and then the geometry of the ellipsoid can tell us which variable might have the biggest influence. Again the problem is that it seems impossible to include into this picture the logical inferences, i.e., the look-ahead. The fundamental problem is that this point of view doesn't seem to deliver a measure or a distance, and thus it cannot be used to compare arbitrary problem instances. Perhaps this restriction can be overcome in the future.

The most promising direction at this time for strengthened distances is presented in Section 7.8, where, based on our general understanding of the interplay between distances and trees, possibilities are discussed to optimise distance functions, even online, so that one can start with a reasonable distance, as presented in the subsequent Subsection 7.7.4.1, and then adapt it to the problem at hand.

7.7.4.1. Maximise the number of new clauses

We start with presenting the current state-of-the-art, which in its pure form is the branching rule "MNC" ("maximise new clauses") used by the `OKsolver`-2002, and from this we make a "rational reconstruction" of earlier rules (in Subsection 7.7.4.2), which can well be understood as approximations of MNC.

MNC for a branching variable v and the residual clause-set F considers the two branches F_0, F_1, where for a standard look-ahead solver we have $F_\varepsilon = \varphi_\varepsilon * F$, where φ_ε is the extension of $\langle v \to \varepsilon \rangle$ by unit-clause propagation, and uses as distance the weighted number of new clauses

$$\mathrm{d}_{\mathrm{MNC}}(F, F') = \sum_{k=2}^{\mathrm{rank}(F)-1} w_k \cdot c_k(F' \setminus F).$$

More precisely, multi-clause-sets should be used here, since typically in practice multiple occurrences of the same new clause are counted, and no contraction with existing clauses takes place. By optimisation on random 3-CNF at the (approximated) threshold, optimal weights could be established as approximatively $w_2 := 1$, $w_3 := 0.2$, $w_4 := 0.05$, $w_5 := 0.01$, $w_6 := 0.003$ and $w_k = 20.45 \cdot 0.2187^k$ for $k \geq 7$, where these weights also work quite well on other problem instances. The "precise" combination of the two distances $\mathrm{d}_{\mathrm{MNC}}(F, F_0), \mathrm{d}_{\mathrm{MNC}}(F, F_1)$ into one number happens via the τ-function, while in practice the product-rule is sufficient (as discussed in Subsection 7.6). After the best variable has been chosen, the first branch is selected as will be discussed in Section 7.9, where in practical applications currently the scheme discussed in Subsection 7.9.1 seems to yield the best results. A few remarks:

1. We have $\mathrm{d}_{\mathrm{MNC}}(F, F_\varepsilon) = 0$ iff $F_\varepsilon \subseteq F$, in which case φ_ε is a "weak autarky" for F, and thus F_ε is satisfiability equivalent to F, so no branching is needed.
2. The more new clauses the better, and the better the shorter they are.
3. New clauses result from falsifying literal occurrences in (old) clauses which are not satisfied — satisfied clauses are ignored by the distance (but will be taken into account when choosing the first branch).
4. When performing look-ahead the behaviour of $\mathrm{d}_{\mathrm{MNC}}$ seems counterintuitive: Consider two variables v, w, where for v we have in both branches many inferred assignments (by unit-clause propagation), but it happens that they both result in just, say, one new clause each; on the other hand, assume that for w in both branches there aren't many inferred assignments (possibly none at all), but more new clauses. Then w will be preferred over v. Such examples can be constructed, but in practice it turned out that attempts at balancing $\mathrm{d}_{\mathrm{MNC}}$, by for example the number of inferred assignments, performed worse in almost all cases (which is also confirmed by the historical development, as outlined in the subsequent subsection).[7]

Instead of clauses also more general "conditions" ("constraints") can be treated, if partial assignments can be applied to them, and instead of new clauses we then need to consider conditions whose domains have been restricted. Since conditions can show much more variety in their behaviour, the problem of the choice of weights for the "new conditions" becomes more pronounced; a first approximation is to replace the length k of clauses by the size of the domain of the condition, but in general one needs to keep in mind that the reason why shorter clauses are preferred over longer ones is that they are *more constrained*, i.e., will easier yield inferred assignments in the future. One can measure the ratio of falsifying assignments for the new conditions (the higher the better), but according to current knowledge extensive experimentation to find good weighting schemes seems unavoidable. One also needs to take into account that while clauses can only yield a single inferred assignment, other constraints can yields more inferred assignments (at once; and potentially also other types of information might be inferred, for example equality between variables). See Subsection 7.10.1 for some examples.

[7]The extreme case of zero new clauses needs to be handled (since distances need to be positive), and for this case we have the autarky-reduction above; in [Kul99b] also the case of an arbitrary number of new clauses is treated via the addition of "autarky clauses" (which actually goes beyond resolution), and the `OKsolver`-2002 contains (deactivated) code for handling the case of exactly one new clause. The future has to show whether this scheme is of value.

7.7.4.2. The historical development (for practical SAT solving)

An early branching scheme is presented in [JW90], and is known as the "Jeroslow-Wang rule". Since it is a rather confused rule, of weak efficiency and mixing up several aspects, we do not further comment on this scheme. However this scheme was at least historically of importance, since it allowed [HV95] to state an improved branching scheme, with improved (though still confused) reasoning: The Jeroslow-Wang rule is replaced by the "two-sided Jeroslow-Wang rule", rejecting the idea that branching schemes are "satisfiability driven", and replacing this paradigm by the "simplification paradigm". We can interprete this rule by the underlying distance $\mathrm{d}_{2\mathrm{JW}}(F, F') := \sum_{k=1}^{\mathrm{rank}(F)-1} w_k \cdot c_k(F' \setminus F)$ where $w_k := 2^{-k}$ (strictly following [HV95] it would be $w_k = 2^{-(k+1)}$, but obviously the factor $\frac{1}{2}$ doesn't matter here). Reduction is unit-clause propagation and pure literal elimination. No look-ahead is used (fitting with the choice of reduction(!)), and as projection the sum is used. The choice of the first branch happens by the Johnson-rule (see Subsection 7.9.2). We see that $\mathrm{d}_{\mathrm{MNC}}$ improves $\mathrm{d}_{2\mathrm{JW}}$ by

1. using the product as projection;
2. discriminating sharper between different clause-length;
3. using r_1-look-ahead (instead of r_0);
4. choosing of the first branch by the Franco-rule (see Subsection 7.9.1).

These shortcomings have been addressed piecewise by later developments as outlined below. Regarding our interpretation, we need to stress that the understanding of the "two-sided Jeroslow-Wang rule" as based on the distance of counting *new* clauses is a rational reconstruction, while the argumentation in [HV95] tries to argue in the direction of "simplification". However this is not really what $\mathrm{d}_{2\mathrm{JW}}$ is aiming at, namely increasing the number of future forced assignments (by unit-clause propagation or stronger means). In [HV95] we find a misleading understanding of the sum $\mathrm{d}_{2\mathrm{JW}}(F, \langle v \to 0\rangle * F) + \mathrm{d}_{2\mathrm{JW}}(F, \langle v \to 1\rangle * F)$ (maximised by the heuristics), which is there understood as estimating the simplification (as a kind of average over both branches), and so the real target of the rule, creating many new short clauses (for both branches(!)), where then the sum only acts as a (bad) projection, remained hidden.

[VT96] improves upon this branching scheme by using the product as projection (incorporated in the branching rule `dsj`), apparently for the first time, while still not using look-ahead. [DABC96] (the `C-SAT` solver) did not yet use the product-projection, but introduced certain aspects of look-ahead (incorporating the first round of unit-clause propagation) and also a steeper decline for the weights w_k (close to the weights used by the `OKsolver`-2002 scheme), namely now $w_k = -\log(1 - (2^{k+1} - 1)^{-2})$, where the quotient $\frac{w_k}{w_{k+1}}$ is monotonically decreasing, with limit 4. In Section 3.1 of [Ouy99] a possible derivation of this rule is offered. A similar approach (but already using the product-projection, and partially using r_2 as reduction), improving the `Satz` solver, is discussed in [Li99]. While these heuristics aimed at "doing a thorough job", there is also the direction of "cheaper heuristics", mostly focusing on efficient implementations. The basic idea is "MOM", i.e., "maximum occurrences in clauses of minimum size", and all these heuristics can be understood as approximations of "maximise the number of new clauses", driven by the kind of available data which happens to be supported by the data structures. The solver `Posit` ([Fre95]) introduced efficient data structures and an efficient implementation of r_2-reduction (which altogether made it very competitive), but regarding heuristics no progress over [VT96] took place. Once the efficiency of r_2-reduction ("failed literal eliminations") for this type of solvers (the "look-ahead solvers") became apparent, the cost of measurements needed to evaluate $\mathrm{d}_{\mathrm{MNC}}$ and variations as well as the cost of look-aheads diminishes (relatively), and the branching scheme as embodied in its pure form by the `OKsolver`-2002 (as discussed in Subsection 7.7.4.1) is now the common standard of look-ahead solvers. The `march`-solvers extended the scheme in various ways (see [Heu08]), considering only a subset of variables for the look-ahead, as pioneered by [LA97], but the core rule has been left unchanged.

7.8. Optimising distance functions

Now we turn to the topic of improving distance functions by evaluating how good they actually predict progress as given by the real branching trees. The possibility of evaluating distances leads to the possibility of actually optimising distances, and the two basic approaches are considered in Subsections 7.8.2, based on the evaluation techniques discussed in Subsection 7.8.1. The subject of optimising distances has been studied in the context of theoretical upper bounds (see Subsection 7.7.3). For practical SAT solving however there seems to be no general attempt (considering arbitrary distances) in the literature. The notion of "adaptive heuristics" as used in [Hv07] is based on a broad view of "heuristics", and actually only concerns the reduction process (see Subsection 7.7.2), by dynamically adapting the set of variables eligible for reduction considerations, thus weakening the reduction by excluding variables, and also adapting the set of variables for a strengthened reduction.

7.8.1. Evaluating distance functions

The "relative" effectiveness of the distance function d used in a solver on one particular instance (with medium-size branching tree) can be gauged (that is, regarding its "relative values") as follows: Compute the distribution of the random variable P^{-1} induced by $\mathfrak{P}_d$ (see Definition 7.4.3) — a good distance function yields a centred distribution (around $\mathrm{E}(\mathrm{P}^{-1}) = \#\mathrm{lvs}(T)$), while a bad distance function is more similar to the uniform distribution.

This evaluation is also computationally not very expensive: At each node along the current path the probabilities of the branches are computed, and once a leaf is found, then the product of these probabilities along the whole path is dumped to a file; viewing the distribution of the random variable P with a statistical tool kit (like R) might reveal interesting phenomena.[8] That we consider only "relative values" of d here means that an "optimal distance" d (with constant

[8]Handling the logarithms of the probabilities is likely advantageous. And one should keep

$\mathrm{P}^{-1} = \#\mathrm{lvs}(T)$) is derived from a canonical distance $\Delta\mu$ with $\mu(v) = \log \#\mathrm{lvs}(T_v)$ (recall Lemma 7.4.9) by multiplying the branching tuple at each inner node with an *individual* positive real number (possibly different for each node).

Care is needed with the interpretation of such evaluations: Comparing different distance functions can be misleading (even on the same tree), since, metaphorically speaking, to make some interesting predictions something needs to be risked, and a thumb heuristic can have a smaller variance then a more powerful heuristic.

A weakness in using the variance for evaluating distance functions is that the absolute values of the distances at different nodes are ignored. The second basic possibility for evaluating distances on (given) trees is to use the upper bound from Theorem 7.4.8, which on the other hand has the disadvantage that only the worst nodes are considered (while the variance includes all nodes).

7.8.2. Minimising the variance or minimising the tau-upper-bound

A numerical value for the quality of the distance function d is given by the variance $\mathrm{Var}(\mathrm{P}^{-1})$ (the smaller the better, and the optimal value is 0); for convenience the standard deviation $\sqrt{\mathrm{Var}(\mathrm{P}^{-1})}$ might be used, and for comparing different trees the normalised standard deviation $\frac{\sqrt{\mathrm{Var}(\mathrm{P}^{-1})}}{\#\mathrm{lvs}(T)}$ is appropriate, however these quantities only serve as a "user interface" in our context, and are not considered furthermore. By Lemma 7.4.4 we have an efficient recursive method for computing the variance, which also does not cause space overhead since no additional storage is required (if we do not want to visually inspect the spread of P^{-1}). If we have given several distance functions $d_1, \dots, d_k$, then we can choose the best one (on average) as the one with minimal $\mathrm{Var}(\mathrm{P}_{d_i}^{-1})$. This method can also be used "on the fly", during the execution of a backtracking solver, with negligible overhead, to dynamically adapt the solver to the problem at hand. Of course, measuring the quality of a distance function by the variance $\mathrm{Var}(\mathrm{P}_d^{-1})$ enables not only comparison of different distance functions, but if d depends on parameter α (possibly a vector), then we can also optimise d_α by minimising $\mathrm{Var}(\mathrm{P}_{d_\alpha}^{-1})$ (for the given tree). A special (interesting) case is where d is given as a convex linear combination of distances $d_1, \dots, d_k$ (that is, $d = \sum_{i=1}^{k} \lambda_i \cdot d_i$ for $\lambda_i \geq 0$ and $\sum_{i=1}^{k} \lambda_i = 1$). Actually, here the d_i do not need to be distances, but could even assume negative values, if only we take care to restrict the λ_i accordingly to avoid non-positive values of d. These optimisations could be also performed "online".

Instead of minimising $\mathrm{Var}(\mathrm{P}_{d_\alpha}^{-1})$ one could consider minimising $\max(\mathrm{P}_{d_\alpha}^{-1}) = \max_{v \in \mathrm{lvs}(T)} \mathrm{P}_{d_\alpha}(v)^{-1}$, i.e., minimising the worst-case upper bound from Lemma 7.4.1. This task is considerably simplified by applying logarithms. However this optimisation is much rougher, since it only cares about getting rid off the most extreme cases (this is the worst-case perspective), which might be weak since the worst case might occur only rarely. So the only real alternative seems to minimise the upper bound $(\max \tau(d))^{\max \Sigma(d)}$, or better $\log((\max \tau(d))^{\max \Sigma(d)}) =$

in mind that on the leaves we do not consider the uniform probability distribution, and so just plotting the "observations " is not meaningful.

$(\max \Sigma(d)) \cdot \log(\max \tau(d))$, which still suffers from considering only the worst case, but has the advantage that scaling of the distances is taken into account.

7.9. The order of branches

After having chosen the branching variable, the next step then is to order the branches, which according to Subsection 7.7.2 is the task of the *branching ordering heuristics* (thus ordering of branches is not integrated into the τ-method, but is an additional step).[9] We extract an approximated "probability" that we have a favourable situation, like finding a satisfying assignment for look-ahead solvers, or "learning a lot" for conflict-driven solvers, and order the branches according to this approximation (in descending order). Unfortunately yet not much theoretically founded is known about clause-learning, and the question here is only how to compute "approximations" of some form of probability that a clause-set F is satisfiable. Such "approximations" are achieved by *satisfiability estimators P*.

For unsatisfiable problem instances the order doesn't matter (without learning), while for satisfiable instances, whatever the branching is, finding the right first branch actually would solve the problem quickly (when ignoring the time for choosing the first branch). While in previous sections often the problem representation was not of importance, and very general satisfiability problems over non-boolean variables could be handled, now the approaches are combinatorially in nature and thus are more specific to (boolean) clause-sets. In Subsections 7.9.1 and 7.9.2 we consider the two schemes currently used in practice[10]. A difficulty is that these two satisfiability estimators for choosing the first branch do not have standard names; for ease of reference we propose the following names:

1. Since apparently the first approach has first been mentioned by John Franco, we call it the "Franco heuristics" or "Franco estimator".
2. The second approach was apparently first mentioned in [Joh74] in the context of "straight-line programs", which is close to our usage, while the "Jeroslow-Wang heuristics" from [JW90] misuses the heuristics to actually choose the branching *variable* (as discussed in Subsection 7.7.4.2), so it seems sensible to call the heuristics the "Johnson heuristics" (or the "Johnson estimator").

7.9.1. The Franco estimator: Considering random clause-sets

This approach, apparently first mentioned in the literature in [GPFW97] (in the context of backtrack-free algorithms for random formulas), considers F as a

[9]In [Hv08] this is called "direction heuristics". We prefer to speak of the ordering of branches since our methods work for arbitrarily wide branchings, and they consider not just the first branch.

[10]The `OKsolver`-2002 uses the first scheme, and on random 3-CNF this scheme appears to be slightly stronger than the second one, an observation already made 1986 by John Franco in the context of (incomplete) "straight-line SAT algorithms" (which do not backtrack but abort in case of a failure).

random element $F \in \Omega(p_1(F), \dots, p_k(F))$ of some probability space Ω depending on parameters p_i. So now we can speak of the probability $P_\Omega(F \in \mathcal{SAT})$ that F is satisfiable. Yet this approach has not been applied in this form, and one does not consider the (complicated) probability $P_\Omega(F \in \mathcal{SAT})$ of "absolute" satisfiability, but instead the probability $P_\Omega(\varphi * F \in \mathcal{SAT})$ that a random (total) assignment φ satisfies a random $F \in \Omega$. Most natural is to consider the constant density model (with mixed densities), that is one considers all clause-sequences F as equally likely which have $n(F)$ many (formal) variables and $c(F)$ many clauses in total, where for clause-length k we have $c_k(F)$ many clauses. Then actually the random assignment φ_0 can be fixed, say to the all-zero assignment, and due to the independence of the clause-choices we have $P(\varphi_0 * F \in \mathcal{SAT}) = \prod_{C \in F}(1 - 2^{-|C|})$. For efficiency reasons the use of the logarithm $L(F) := \log \prod_{C \in F}(1 - 2^{-|C|}) = \sum_{C \in F} \log(1 - 2^{-|C|})$ is preferable, where furthermore the factors $\log(1 - 2^{-k})$ for relevant clause-lengths $k = 1, 2, 3, \dots$ can be precomputed. So the first approach yields the rule to order a given branching $(F_1, \dots, F_m)$ by descending $L(F_i) = \sum_{k \in \mathbb{N}_0} c_k(F_i) \cdot \log(1 - 2^{-k})$.

7.9.2. The Johnson estimator: Considering random assignments

The second approach considers the specific F equipped with the probability space of all total assignments, and the task is to approximate the probability $\#\mathrm{sat}(F)/2^{n(F)}$ that a random assignments satisfies F, where $\#\mathrm{sat}(F)$ is the number of satisfying (total) assignment. Since for conjunctive normal forms falsifying assignments are easier to handle than satisfying assignments, we switch to the consideration of the probability $P_0(F) := \#\mathrm{usat}(F)/2^{n(F)} = 1 - \#\mathrm{sat}(F)/2^{n(F)}$ that a total assignment falsifies F (where $\#\mathrm{usat}(F)$ is the number of total assignments falsifying F). We have the obvious upper bound $P_0(F) \le \#\mathrm{usat}(F)/2^{n(F)} \le P_0^1(F) := \sum_{C \in F} 2^{-|C|}$, which is derived from considering the case that no total assignment falsifies two (different) clauses of F at the same time. The lower index "0" in $P_0^1(F)$ shall remind of "unsatisfiability", while the upper index indicates that it is the first approximation given by the "inclusion-exclusion" scheme. Using P_0^1 as approximation to P_0, we obtain the rule to order a branching $(F_1, \dots, F_m)$ by ascending $\sum_{k \in \mathbb{N}_0} c_k(F_i) \cdot 2^{-k}$. We see that this method in principle is very similar to the first method, in both cases one minimises (for the first branch) the weighted number $c^w(F) = \sum_{k=0}^{\infty} w(k) \cdot c_k(F)$ of clauses of F, where the weights $w(k)$ depend only on the length k of the clauses. The only difference between these two methods are the weights chosen: for the first method we have $w_1(k) = -\log(1 - 2^{-k})$, for the second $w_2(k) = 2^{-k}$. Asymptotically we have $\lim_{k \to \infty} \frac{w_1(k)}{w_2(k)} = 1$, since for small x we have $-\log(1 - x) \approx x$. So the second method can also be understood as an approximation of the first method.

Another understanding of this heuristic comes from [Joh74]: $P_0^1(F)$ is the (exact) expected number of falsified clauses for a random assignment (this follows immediately by linearity of expectation). We remark that thus by Markov's inequality we obtain again $P_0(F) \le P_0^1(F)$. And finally we mention that if $P_0^1(F) < 1$ holds, then F is satisfiable, and the Johnson heuristic, used either without look-ahead or with look-ahead involving unit-clause-propagation, will actually find a satisfying assignment without backtracking, since in case of $P_0^1(F) < 1$ for every variable $v \in \mathrm{var}(F)$ there exists $\varepsilon \in \{0,1\}$ with $P_0^1(\langle v \to \varepsilon \rangle * F) < 1$. We also see that using the Johnson heuristic without backtracking and without look-ahead yields an assignment which falsifies at most $P_0^1(F)$ many clauses of F. For fixed uniform clause-length k we get at most $P_0^1(F) = c(F) \cdot 2^{-k}$ falsified clauses, that is at least $c(F) \cdot (1 - 2^{-k})$ satisfied clauses, which actually has been shown in [Hås01] to be the optimal approximation factor $(\frac{1}{1-2^{-k}})$ for the maximal number of satisfied clauses which can be achieved in polynomial time (unless P = NP).

7.9.3. Alternative points of view

It appears to be reasonable that when comparing different branchings (for SAT this typically means different branching variables) where one branching has a branch with a very high associated probability of satisfiability, that then we take the satisfiability-aspect more important than the reduction-aspect, since we could be quite sure here. Yet it seems that due to the crudeness of the current schemes such considerations are not very successful with the methods discussed above, however they are applied in the context of a different paradigm, which does not use approximated satisfiability probabilities of problem instances for the ordering of branches, but uses approximations of *marginal probabilities* for single variables as follows: Consider a satisfiable problem instance F (for unsatisfiable instances the order of branches does not matter in our context) and a variable $v \in \mathrm{var}(F)$. If we can compute reasonable approximations $\tilde{p}_v(\varepsilon)$ for values $\varepsilon \in \{0,1\}$ of the ratio $p_v(\varepsilon)$ of satisfying (total) assignments f with $f(v) = \varepsilon$, then we choose first the branch ε with higher $\tilde{p}_v(\varepsilon)$. The main structural problem of this approach is that it focuses on single variables and cannot take further inferences into account, while when having a satisfiability probability estimator $P(F)$ at hand, then we can improve the accuracy of the approximation by not just considering $P(\langle v \to \varepsilon \rangle * F)$ but applying further inferences to $\langle v \to \varepsilon \rangle * F$. However especially for random problems this approach shows considerable success, and so we conclude this section by some pointers to the relevant literature.[11]

The basic algorithm for computing the marginalised number of satisfying assignments (i.e., conditional on setting a variable to some given value) is the "sum-product algorithm" (see [KFL01]), also known as "belief propagation". This algorithm is exact if the "factor graph", better known to the SAT community as "clause-variable graph" (for clause-sets; a bipartite graph with variables and clauses as the two parts), is a tree. Considering the variable-interaction graph $\mathrm{vig}(F)$ (nodes are variables, joined by an edge if occurring together in some constraint), in [KDG04] approximations $p_v^k(\varepsilon)$ of $p_v(\varepsilon)$ for a parameter $k \in \mathbb{N}_0$

[11]Another problem with this approach is that for good predictions variables v with "strong bias", i.e., with large $|\tilde{p}_v(0) - \tilde{p}_v(1)|$ are preferred, which interferes with the branching heuristics (for example on unsatisfiable instances preferring such variables seems rather senseless). This is somewhat similar to preferring a variable v with one especially high value $P(\langle v \to \varepsilon \rangle * F)$, but now the problem is even more pronounced, since a "confident" estimation $\tilde{p}_v(\varepsilon)$ requires $\tilde{p}_v(1 - \varepsilon)$ to be low, and such biased variables might not exist (even when using precise values).

are studied (for the purpose of value ordering) where the approximation is precise if $k \leq \mathrm{tw}(\mathrm{vig}(F))$, the treewidth of the variable-interaction graph. A recent enhancement of belief propagation is "survey propagation"; see [BMZ05] for the original algorithm, and [Man06, HM06] for further generalisations.

Finally, to conclude this section on the order of branches, some aspect is worth to mention which indicates that the order of branches should also take the (expected) complexity of the branches into account. Say we have a branching variable v, where branch $v \to 0$ has approximated SAT probability 0.7 and expected run time $1000s$, while branch $r \to 1$ has approximated SAT probability 0.3 and expected run time $1s$. Then obviously branch $v \to 1$ should be tried first. Apparently this approach has not been transformed yet into something really workable, likely for the same reason as with the "rational branching rule" mentioned in Subsection 7.2.1, namely that the estimations for running times are far too rough, but it explains the erratic success of methods for choosing the branching order according to ascending expected complexity (easiest problem first) in practice on selected benchmarks.

7.10. Beyond clause-sets

The general theory of branching heuristics, developed in Section 7.2 to Section 7.6, is applicable to any backtracking algorithm, including constraint satisfaction problems (with non-boolean variables), and also the methods discussed in Section 7.8 are applicable in general. However the special heuristics discussed in Section 7.7 and Section 7.9 focused on SAT for boolean CNF. In this section we give an outlook on heuristics using more general "constraints" than clauses, considering "generalised clauses" in Subsection 7.10.1 (for examples BDD's), which support generic SAT solving (via the application of partial assignments), and considering representations from constraint programming in Subsection 7.10.2.

7.10.1. Stronger inference due to more powerful "clauses"

Staying with boolean variables, a natural candidate for strengthened clauses are OBDDs. [FKS$^+$04] is a good example for this direction, which also contains an overview on other approaches. Actually, OBDDs are only a stepping stone for [FKS$^+$04], and the form of of generalised clauses actually used are "OBBDs on steroids", called "smurf's", which store for every partial assignment (in the respective scope) all inferred assignments. In accordance with the general scheme MNC from Subsection 7.7.4.1, but without (i.e., with trivial) look-ahead[12], [FKS$^+$04] define for each smurf a weight after the branching assignment, which reflects the reduction in the number of variables of the smurf due to inferred assignments, directly and with exponential decay also for the possible futures. More lightweight approaches include equivalence reasoning; see [Li03] for heuristical approaches to include the strengthened reasoning efficiently into the look-ahead of the heuristics.

[12]likely due to the fact that standard clause-learning is used, whose standard data structures are incompatible with stronger reductions and look-aheads

7.10.2. Branching schemes for constraint satisfaction

While the previous Subsection 7.10.1 considered "generalised SAT" in the sense that, though more general "constraints" are used, they are very "structured" w.r.t. allowing efficient inspection of the effects of (arbitrary) partial assignments. The field of constraint satisfaction on the other side tends to take a black-box point of view of constraints. The theory developed in this chapter yields a straightforward canonical basic heuristics for this environment, and we present these considerations below. First however some general comments on the discussion of branching heuristics for constraint satisfaction problems (CSPs) in [van06].

The *branching strategy* (see Section 4.2 in [van06]) selects the way in which to split the problem. For SAT solving binary branching on a variable is absolutely dominant (only in theoretical investigations more complex branchings are used), and thus for practical SAT solving the problem of the choice of branching strategy does not exist (at this time): An essential feature of SAT solving (compared for example with CSP solving) is that problems are "shredded" into tiny pieces so that the "global intelligence" of a SAT solver can be applied, and only "micro decisions" are made (via *boolean* variables), always on the outlook for a better opportunity (which might arise later, while by a more complex branching we might have made choices too early). Translating a problem with non-boolean variables to SAT (with boolean variables) typically increases the search space. On the other hand, [MH05] showed that this increased search space also contains better branching possibilities: A d-way branching for a variable v with domain D_v of size d (i.e., $v = \varepsilon_1$, $v = \varepsilon_2$, ..., $v = \varepsilon_d$ for $D_v = \{\varepsilon_1, \dots, \varepsilon_d\}$) can always be efficiently simulated by a 2-way branching (corresponding to $v = \varepsilon, v \neq \varepsilon$ for some $\varepsilon \in D_v$), but not vice versa.

The *variable ordering heuristics* (see Section 4.6.1 in [van06]) is responsible for choosing the branching variable (given that only branching on a single variable is considered), which for SAT solving is typically just called the "branching heuristics". The general tendency seems to be also to choose a variable minimising the expected workload, but surprisingly the integration of the information on the single branches (for the different values in the domain of the variable) into one single number, i.e., the task of the evaluation projection, has apparently never been systematically considered, and consequently the handling of projection is weak. Thus, though 2-way branching can have an edge, as mentioned above, there seems to be some scope of improvement for existing d-way branching (and other schemes yielding non-binary branching) by using the τ-projection (which plays out its strength in a context where various branching widths occur!).

The *value ordering heuristics* (see Section 4.6.2 in [van06]) is responsible for choosing the order of the branches (compare Section 7.9 in this chapter). In (standard, i.e., binary) SAT branching this just means the choice of the first branch.

Now to the most basic measures and distances. The basic estimation for problem complexity for boolean problems F is $2^{n(F)}$, the size of the search tree, from which we derive the measure $\mu(F) = n(F) = \log_2 2^{n(F)}$ (motivated by Lemma

7.4.9). As already mentioned at several places, using the distance Δn is rather weak for general SAT solving. However this measure can be strengthened by either strengthening the look-ahead, or by differentiating more between variables. The latter is possible for CSP problems, where variables v have domains D_v, and then $\mu(F) = \log \prod_{v \in \mathrm{var}(F)} |D_v| = \sum_{v \in \mathrm{var}(F)} \log(|D_v|)$ becomes a "more informed" measure of problem complexity (note that $\mu(F) = n(F)$ if all variables are binary). When considering arbitrary branching width, usage of the τ-function becomes compulsory, and comparing different branchings $F \rightsquigarrow (F_1^i, \ldots, F_{p_i}^i)$ is then done by choosing the branching i with minimal $\tau(\Delta\mu(F, F_1^i), \ldots, \Delta\mu(F, F_{p_i}^i))$. When combined with look-ahead (recall, this always refers to the branching heuristics (alone), that is, the F_j^i are better approximations of the reduced formula computed when actually choosing this branch), this yields decent basic performance. However if more information on the constraints is available, then the approach as discussed at the end of Subsection 7.7.4.1 likely will yield stronger results.

7.11. Conclusion and outlook

Regarding branching heuristics (the main topic of this chapter), one could say that historically the first phase has been finished by now, laying the foundations by constructing basic heuristics, as outlined in Subsection 7.7.4, and developing the basic theory as outlined in Sections 7.3 to 7.5. Now a much stronger emphasise should be put on precise quantitative analysis, as discussed in Section 7.8 on optimising (and measuring) the quality of heuristics, including the consideration of specialised branching projections as discussed in Section 7.6. Case studies on classes of problems should be valuable here. When saying that for branching heuristics the initial phase of theory building and systematic experimentation is finished, then actually this can only be said about finding the branchings, while regarding ordering the branches (as surveyed in Section 7.9) we are lacking a good theoretical foundation, and likely also much further experimentation is needed.

The general theory (on finding good branchings and ordering them) is applicable also beyond clause-sets (as discussed in Section 7.10), and while for clause-sets we have at least reasonable intuitions what might be good measures (distances) and ordering heuristics, here the situation is much more complicated. Combining the theory developed in this chapter with the different approaches from the field of constraint satisfaction (which in my opinion lacks the general theory, but has a lot to say on specific classes of constraints) should be a good starting point.

Finally, a major enterprise lies still ahead of us: theoretical foundations for heuristics for conflict-driven SAT solvers. In the introductions to Sections 7.7 and 7.9 we made a few remarks on conflict-driven solvers, but mostly we did not cover them in this chapter due to their non-tree-based approach and the lack of a theoretical foundation. See Chapter 4 of this handbook for more information on these solvers and the underlying ideas, where the basic references are as follows: With the solver `GRASP` ([MSS99]) the basic ideas got rolling, then `Chaff` ([MMZ+01, ZMMM01]) was considered by many as a breakthrough, while further progress was obtained by the solvers `BerkMin` ([GN02]) and `MiniSat` ([ES04]).

References

[BF81] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume I.* William Kaufmann, Inc., 1981. ISBN 0-86576-005-5.

[BKD$^+$04] Per Bjesse, James Kukula, Robert Damiano, Ted Stanion, and Yunshan Zhu. Guiding SAT diagnosis with tree decompositions. In Giunchiglia and Tacchella [GT04], pages 315–329. ISBN 3-540-20851-8.

[BKS04] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[BMZ05] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, March 2005.

[Bul03] P. S. Bullen. *Handbook of Means and Their Inequalities*, volume 560 of *Mathematics and Its Applications.* Kluwer Academic Publishers, 2003. ISBN 1-4020-1522-4.

[CFR05] Paul Cull, Mary Flahive, and Robby Robson. *Difference Equations: From Rabbits to Chaos.* Undergraduate Texts in Mathematics. Springer, 2005. ISBN 0-387-23234-6.

[DABC96] Olivier Dubois, P. Andre, Y. Boufkhad, and C. Carlier. SAT versus UNSAT. In Johnson and Trick [JT96], pages 415–436. The Second DIMACS Challenge.

[Epp06] David Eppstein. Quasiconvex analysis of multivariate recurrence equations for backtracking algorithms. *ACM Transactions on Algorithms*, 2(4):492–509, October 2006.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Giunchiglia and Tacchella [GT04], pages 502–518. ISBN 3-540-20851-8.

[FGK05] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 87:47–77, October 2005.

[FKS$^+$04] John Franco, Michal Kouril, John Schlipf, Sean Weaver, Michael Dransfield, and W. Mark Vanfleet. Function-complete lookahead in support of efficient SAT search heuristics. *Journal of Universal Computer Science*, 10(12):1655–1692, 2004.

[Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[GN02] Evguenii I. Goldberg and Yakov Novikov. Berkmin: A fast and robust Sat-solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002.

[GPFW97] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfability (SAT) problem: A survey. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem:*

Theory and Applications (DIMACS Workshop March 11-13, 1996), volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–151. American Mathematical Society, 1997. ISBN 0-8218-0479-0.

[GT04] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing 2003*, volume 2919 of *Lecture Notes in Computer Science*, Berlin, 2004. Springer. ISBN 3-540-20851-8.

[Hås01] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, July 2001.

[Heu08] Marijn J. H. Heule. *SMART solving: Tools and techniques for satisfiability solvers.* PhD thesis, Technische Universiteit Delft, 2008. ISBN 978-90-9022877-8.

[HLP99] G. H. Hardy, J. E. Littlewood, and G. Pólya. *Inequalities.* Cambridge Mathematical Library. Cambridge University Press, second edition, 1999. ISBN 0-521-35880-9; reprint of the second edition 1952.

[HM06] Eric I. Hsu and Sheila A. McIlraith. Characterizing propagation methods for boolean satisfiability. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 325–338. Springer, 2006. ISBN 3-540-37206-7.

[HUL04] Jean-Baptiste Hiriart-Urruty and Claude Lemaréchal. *Fundamentals of Convex Analysis.* Grundlehren (text editions). Springer, 2004. ISBN 3-540-42205-6; second corrected printing.

[HV95] John N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[Hv07] Marijn J. H. Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In Joao P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 258–271. Springer, 2007. ISBN 978-3-540-72787-3.

[Hv08] Marijn J. H. Heule and Hans van Maaren. Whose side are you on? Finding solutions in a biased search-tree. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:117–148, 2008.

[Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, December 1974.

[JT96] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996. The Second DIMACS Challenge.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[KDG04] Kalev Kask, Rina Dechter, and Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Principles and Practice* *of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science (LNCS)*, pages 317–331, 2004.

[KFL01] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, February 2001.

[KL97] Oliver Kullmann and Horst Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Preprint, 82 pages; the ps-file can be obtained at `http://cs.swan.ac.uk/~csoliver/`, January 1997.

[KL98] Oliver Kullmann and Horst Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages; the ps-file can be obtained from `http://cs.swan.ac.uk/~csoliver/`, December 1998.

[Knu75] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, January 1975.

[Kul92] Oliver Kullmann. Obere und untere Schranken für die Komplexität von aussagenlogischen Resolutionsbeweisen und Klassen von SAT-Algorithmen. Master's thesis, Johann Wolfgang Goethe-Universität Frankfurt am Main, April 1992. (Upper and lower bounds for the complexity of propositional resolution proofs and classes of SAT algorithms (in German); Diplomarbeit am Fachbereich Mathematik).

[Kul98] Oliver Kullmann. Heuristics for SAT algorithms: A systematic study. In *SAT'98*, March 1998. Extended abstract for the Second workshop on the satisfiability problem, May 10 - 14, 1998, Eringerfeld, Germany.

[Kul99a] Oliver Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF's based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), October 1999.

[Kul99b] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, July 1999.

[Kul02] Oliver Kullmann. Investigating the behaviour of a SAT solver on random formulas. Technical Report CSR 23-2002, Swansea University, Computer Science Report Series (available from `http://www-compsci.swan.ac.uk/reports/2002.html`), October 2002.

[Kul03] Oliver Kullmann. Lean clause-sets: Generalizations of minimally unsatisfiable clause-sets. *Discrete Applied Mathematics*, 130:209–249, 2003.

[Kul08a] Oliver Kullmann. Fundaments of branching heuristics: Theory and examples. Technical Report CSR 7-2008, Swansea University, Computer Science Report Series (`http://www.swan.ac.uk/compsci/research/reports/2008/`), April 2008.

[Kul08b] Oliver Kullmann. Present and future of practical SAT solving. In

Nadia Creignou, Phokion Kolaitis, and Heribert Vollmer, editors, *Complexity of Constraints*, volume 5250 of *Lecture Notes in Computer Science (LNCS)*, pages 283–319. Springer, 2008.

[LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371. Morgan Kaufmann Publishers, 1997.

[Li99] Chu Min Li. A constraint-based approach to narrow search trees for satisfiability. *Information Processing Letters*, 71(2):75–80, 1999.

[Li03] Chu Min Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130:251–276, 2003.

[Luc84] Horst Luckhardt. Obere Komplexitätsschranken für TAUT-Entscheidungen. In *Frege Conference 1984, Schwerin*, pages 331–337. Akademie-Verlag Berlin, 1984.

[Man06] Elitza Nikolaeva Maneva. *Belief propagation algorithms for constraint satisfaction problems.* PhD thesis, University of California, Berkeley, 2006.

[MH05] David G. Mitchell and Joey Hwang. 2-way vs. d-way branching for CSP. In Peter van Beek, editor, *Principles and Practice of Constraint Programming — CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2005.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[MS85] B. Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[MSS99] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[Ouy99] Ming Ouyang. *Implementations of the DPLL algorithm.* PhD thesis, Graduate School—New Brunswick; Rutgers, The State University of New Jersey, 1999.

[Ren92] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. *Journal of Symbolic Computation*, 13:255–352, 1992.

[Sch92] Ingo Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. In *Selected papers from Computer Science Logic '92*, volume 702 of *Lecture Notes Computer Science*, pages 379–394, 1992.

[van06] Peter van Beek. Backtracking search algorithms. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 4, pages 85–134. Elsevier, 2006. ISBN 0-444-52726-5.

[VT96] Allen Van Gelder and Yumi K. Tsuji. Satisfiability testing with more reasoning and less guessing. In Johnson and Trick [JT96], pages 559–586. The Second DIMACS Challenge.

[vW00] Hans van Maaren and Joost Warners. Solving satisfiability problems using elliptic approximations — effective branching rules. *Discrete Applied Mathematics*, 107:241–259, 2000.

[Wah07] Magnus Wahlström. *Algorithms, Measures and Upper Bounds for Satisfiability and Related Problems.* PhD thesis, Linköpings universitet, Department of Computer and Information Science, SE-581 83 Linköping, Sweden, 2007. ISBN 978-91-85715-55-8.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285. IEEE Press, 2001.