

Searching Trees: an Essay

Henning Fernau and Daniel Raible

Univ.Trier, FB 4—Abteilung Informatik, 54286 Trier, Germany
{fernau,raible}@uni-trier.de

Abstract. We are reviewing recent advances in the run time analysis of search tree algorithms, including indications to open problems. In doing so, we also try to cover the historical dimensions of this topic.

1 Introduction

Search trees are a basic tool for solving combinatorial problems. The underlying idea is quite simple: decompose a given problem into finer and finer partial problems (applying a suitable branching operation) such that the generated partial problems together solve the original problem. Hence, they have been investigated from various points of views for about 50 years, so nearly through all the history of computer science and its mathematics.

Despite of its long history, there are (to our knowledge) only few attempts to develop a unifying view on this topic, apart from some quite old papers, rarely quoted these days, as discussed below. This essay can be seen as a quest to resume a generic research on search trees, with the specific aim to bring the sometimes very successful applications of search trees in practice closer to the theoretical (worst case) analysis. We believe there are many yet unsolved questions.

2 Historical notes on search trees

Conceptual frameworks for search trees. A nice framework on search trees has been developed by Ibaraki in 1978 [30]; actually, that paper sums up many early works. That paper presented a framework that allows to justify the correctness of branch-and-bound procedures. To this end, combinatorial optimization problems are described in the form of *discrete decision processes* (ddp). This formalization actually goes back to Karp and Held [33]. Branch-and-bound, from its very name, implies (in the case of minimization problems) the existence of a lower-bound test. In fact, this was in the focus of earlier formalizations of the principle as, e.g., the one by Mitten [40] that also sums up and reviews still earlier results from the sixties. In those days, many seemingly different names were in use for search tree algorithms; for example, the *backtrack programming* framework of Golomb and Baumert [22] gives another formalization of search tree algorithms. This allows to prune the search tree at node n , assuming it is known that better solutions are already known, compared to the best possible solution that might be derived upon further branching starting out from n . Good pruning hence necessitates

good estimates on the values that could be obtained from n onwards (without necessarily expanding n), which is a (practical and mathematical) problem on its own right. Besides this test, Ibaraki considers two more types of tests (that are quite related among themselves again): *dominance tests* and *equivalence tests*. Roughly speaking, due to a dominance test (considered in details in [29]), we can prune branches in the search tree, since we are sure that better solutions can be found in other branches. Equivalence tests might prove two nodes to be equivalent in the sense that they yield solutions of the same quality, so that only one of the nodes need to be expanded; which one could be the matter of choice due to other criteria. The mentioned early papers focus on the important issue of correctness of the described methods (and how to ensure correctness in concrete circumstances). Typical conceptual questions include: What are the logical relations between abstract properties of discrete decision processes? or: What kind of properties of dominance relations are needed to ensure correctness of the implied pruning strategy?

Artificial Intelligence. A non-negligible part of the literature on Artificial Intelligence deals with search spaces since the very early days. In this sense, search trees play a prominent role in that area, as well. In particular, this remark is true when considering game strategies (like how to play chess with a computer). There again, many (mostly heuristic) decisions have to be made to find a successful path in the underlying implicit configuration tree. We only mention that there are quite a lot of nice internet resources available, like <http://www.cs.ualberta.ca/~aixplore/>. Also in that area of expertise, search trees in the more technical sense of this essay have been examined. For example, Reiter's *theory of diagnosis* is based upon so-called Hitting Set Trees, see [45].

Specific communities have worked a lot to optimize and analyze their search tree algorithms. Nice tools (and generalizable methodologies) have been developed there. For example, the search tree analysis of Kullmann [36] within the Satisfiability Community has been a blueprint of similar approaches elsewhere; it could be seen as one of the fathers of the measure-and-conquer paradigm discussed below. Within the integer linear programming (ILP), and more general, the mathematical programming community, branch-and-bound has been very successfully combined with cutting (hyper-)planes and similar approaches, leading to the so-called branch-and-cut paradigm.¹ Branch-and-cut (and in particular, further refinements like branch-and-cut-price, see [32]) has been a very successful paradigm in industrial practice for solving ILPs, up to the point that solving these (NP-hard) problems is deemed to be practically feasible. However, rather simple examples exist that show that a non-educated use of this approach will lead to quite bad running times, see [31]. The arguably best modern book on this topic [1] focuses on the Traveling Salesman Problem, which is a sort of standard testbed for this approach, starting out from the very origins of branch-and-cut [26]. Another community on its own is dealing with binary decision diagrams [8]; we are not aware of any specific search tree analysis in that area.

¹ We gratefully acknowledge discussions on ILP techniques with Frauke Liers.

3 Estimating running times

Most of the theory papers on search trees up to the seventies focus on conceptual properties of search tree algorithms. Only few papers (to our knowledge), e.g., [27,28,34], try to attack the efficiency question of search tree algorithms from an abstract point of view. Interestingly, one paper [28] shows that even in the average case, exponential growth of branch-and-bound algorithms are not avoidable. It is not so clear how later research in the Artificial Intelligence Community on backtracking programs fits into this line of research, see [38].

Run-time estimates of exponential-time algorithms (as being typical for search tree algorithms) have only relatively recently found renewed interest, obviously initiated by the steadily growing interest in parameterized complexity theory and parameterized (and exact exponential-time) algorithms. Yet, the basic knowledge in this area is not very new. This is possibly best exemplified by the textbook of Mehlhorn [39]. There, to our knowledge, the very first parameterized algorithm for the vertex cover problem was given, together with its analysis, much predating the advent of parameterized algorithmics. This algorithm is quite simple: if any edge $e = \{v_1, v_2\}$ remains in the graph, produce two branches in the search tree, one putting v_1 into the (partial) cover and the other one putting v_2 into the cover. In both cases, the parameter k upperbounding the cover size is decremented, and we consider $G - v_i$ instead of G in the recursive calls. Given a graph G together with an upperbound k on the cover size, such a search tree algorithm produces a binary search tree of height at most k ; so in the worst case, its size (the number of leaves) is at most 2^k . Namely, if $T(k)$ denotes the number of leaves in a search tree of height k , the described recursion implies $T(k) \leq 2T(k-1)$, which (together with the anchor $T(0) = 1$) yields $T(k) \leq 2^k$.

This reasoning generalizes when we obtain recurrences of the form:

$$T(k) \leq \alpha_1 T(k-1) + \alpha_2 T(k-2) + \dots + \alpha_\ell T(k-\ell) \quad (1)$$

for the size $T(k)$ of the search tree (which can be measured in terms of the number of leaves of the search tree, since that number basically determines the running time of a search tree based algorithm).

More specifically, α_i is a natural number that indicates that in α_i of the $\sum_j \alpha_j$ overall branches of the algorithm, the parameter value k got decreased by i . Notice that, whenever $\ell = 1$, it is quite easy to find an estimate for $T(k)$, namely α_1^k . A recipe for the more general case is contained in Alg. 1. Why does that algorithm work correctly? Please observe that in the simplest case (when $\ell = 1$), the algorithm does what could be expected. We only mention here that

$$p(x) = x^\ell - \alpha_1 x^{\ell-1} - \dots - \alpha_\ell x^0$$

is also sometimes called the *characteristic polynomial* of the recurrence given by Eq. 1, and the base c of the exponential function that Alg. 1 returns is called the *branching number* of this recurrence. Due to the structure of the characteristic polynomial, c is the only positive root.

Algorithm 1 Simple time analysis for search tree algorithms, called ST-simple

Require: a list $\alpha_1, \dots, \alpha_\ell$ of nonnegative integers, the coefficients of inequality (1)

Ensure: a tight estimate c^k upperbounding $T(k)$

Consider inequality (1) as equation:

$$T(k) = \alpha_1 T(k-1) + \alpha_2 T(k-2) + \dots + \alpha_\ell T(k-\ell)$$

Replace $T(k-j)$ by x^{k-j} , where x is still an unknown to be determined.

Divide the equation by $x^{k-\ell}$.

{This leaves a polynomial $p(x)$ of degree ℓ .}

Determine the largest positive real zero (i.e., root) c of $p(x)$.

return c^k .

Alternatively, such a recursion can be also written in the form

$$T(k) \leq T(k-a_1) + T(k-a_2) + \dots + T(k-a_r). \quad (2)$$

Then, (a_1, \dots, a_r) is also called the *branching vector* of the recurrence.

As detailed in [23, pp. 326ff.], a general solution of an equation

$$T(k) = \alpha_1 T(k-1) + \alpha_2 T(k-2) + \dots + \alpha_\ell T(k-\ell)$$

(with suitable initial conditions) takes the form

$$T(k) = f_1(k)\rho_1^k + \dots + f_\ell(k)\rho_\ell^k,$$

where the ρ_i are the distinct roots of the characteristic polynomial of that recurrence, and the f_i are polynomials (whose degree corresponds to the degree of the roots (minus one)). As regards asymptotics, we can conclude $T(k) \in \mathcal{O}^*(\rho_1^k)$, where ρ_1 is the dominant root.

The exact mathematical reasons can be found in the theory of polynomial roots, as detailed in [23,13,24,36]. It is of course also possible to check the validity of the approach by showing that $T(k) \leq \rho^k$ for the obtained solution ρ by a simple mathematical induction argument.

Due to case distinctions that will play a key role for designing refined search tree algorithms, the recurrences often take the form

$$T(k) \leq \max\{f_1(k), \dots, f_r(k)\},$$

where each of the $f_i(k)$ is of the form

$$f_i(k) = \alpha_{i,1}T(k-1) + \alpha_{i,2}T(k-2) + \dots + \alpha_{i,\ell}T(k-\ell).$$

Such a recurrence can be solved by r invocations of Alg. 1, each time solving $T(k) \leq f_i(k)$. This way, we get r upperbounds $T(k) \leq c_i^k$. Choosing $c = \max\{c_1, \dots, c_r\}$ is then a suitable upper bound.

Eq. (1) somehow suggests that the entities a_j that are subtracted from k in the terms $T(k-a_j)$ in Eq. (2) are natural numbers. However, this need not be the

case, even in the case that the branching process itself suggests this, e.g., taking vertices into the cover to be constructed. How do such situations arise? Possibly, during the branching process we produce situations that appear to be more favorable than the current situation. Hence, we could argue that we take a certain credit on this future situation, this way balancing the current (bad) situation with the future (better) one. Interestingly, this approach immediately leads to another optimization problem: How to choose the mentioned credits to get a good estimate on the search tree size? We will describe this issue in more detail below in a separate section. This sort of generalization is the backbone of the search tree analysis in so-called exact exponential algorithms, where the aim is, say in graph algorithms, to develop non-trivial algorithms for hard combinatorial graph problems with run-times estimated in terms of n (number of vertices) or sometimes m (number of edges). One typical scenario where this approach works is a situation where the problem allows for nice branches as long as large-degree vertices are contained in the graph, as well as for nice branches if all vertices have small degree, assuming that branching recursively generates new instances with degrees smaller than before, see [16,17,44,46]

An alternative generalization is the following one: We arrive at systems of equations. In its most general form, these will again include maximum operators that can be treated as explained above. We are left with solving systems like

$$T^\ell(k) \leq \max_{i=1}^r f_i^\ell(k),$$

where each of the $f_i^\ell(k)$ ($1 \leq i, \ell \leq r$) is of the form

$$f_i^\ell(k) = \alpha_{i,\ell,1} T^i(k-1) + \alpha_{i,\ell,2} T^i(k-2) + \dots + \alpha_{i,\ell,q_\ell} T^i(k-q_\ell).$$

This approach was successfully applied to problems related to HITTING SET, see [9,10,11]. In the case of 3-HITTING SET, the *auxiliary parameter* ℓ counts how many hyperedges of small size have been generated, since branching on them is favorable. We have seen in our examples that an upper bound to the general situation, i.e., a bound on $T(k) = T^0(k)$, takes again the form c^k . Moreover, the other entities $T^\ell(k)$ are upperbounded by $\beta_\ell c^k$ for suitable $\beta_\ell \in (0, 1)$. Now, if we replace $T^\ell(j)$ by $\beta_\ell T(j)$ in the derived inequality system, we are (after dividing inequalities with left-hand side $T^\ell(k)$ by β_k) back to our standard form discussed above. All inequalities involved now take the form: $T(k) \leq \sum_{j \geq 0} \gamma_j T(k-j)$. Notice that $j = 0$ is feasible as long as $\gamma_0 < 1$. Namely, assuming again an upperbound c^k on $T(k)$, the term $\gamma_0 T(k)$ on the right-hand side can be re-interpreted as $T(k + \log_c(\gamma_0))$. Due to $\gamma_0 < 1$, the logarithmic term is negative, so that the parameter is actually reduced. In fact, the system of inequalities of the form

$$T(k) \leq T(k + \log_c(\gamma_0)) + \sum_{j \geq 1} \gamma_j T(k-j)$$

would yield the same solution. In fact, terms like $\gamma T(k)$ on the right-hand side can be interpreted as a case not yielding direct improvement / reduction of the parameter budget, but the knowledge that the overall search tree size is shrunk by factor γ .

Algorithm 2 A simple algorithm for DOMINATING SET

- 1: **if** possible choose a $v \in \text{BLND}$ such that $|N(v) \cap (\text{BLND} \cup \text{INND})| \geq 2$. **then**
 - 2: Binary branch on v (i.e., set v active in one branch, inactive in the other)
 - 3: **else if** possible choose a $v \in \text{BLDO}$ such that $|N(v) \cap (\text{BLND} \cup \text{INND})| \geq 3$. **then**
 - 4: Binary branch on v .
 - 5: **else**
 - 6: Solve the remaining instance in polynomial time using an EDGE COVER algorithm.
-

4 Measure-and-Conquer

In this separate section, we will discuss one of the most successful approaches to run-time estimation of search trees developed in recent years. With this technique it was possible to prove run time upperbounds $\mathcal{O}^*(c^n)$ with $c < 2$ for several hard vertex selection problems. Among these problems (where for years nothing better than the trivial 2^n -algorithm was known) are many variants of DOMINATING SET [16] like CONNECTED or POWER DOMINATING SET [17,44] and FEEDBACK VERTEX SET [14]. The methodology resulted in simplifying algorithms (INDEPENDENT SET [18]) and in speeding up existent non-trivial ones (DOMINATING SET [16,46], INDEPENDENT DOMINATING SET [21] and MAX-2-SAT [43]). This approach also served for algorithmically proving upper bounds on the number of minimal dominating [19] and feedback vertex sets [14].

In this approach, the complexity of an algorithm is not analyzed with respect to $n = |V|$ (or $m = |E|$) for a graph instance $G = (V, E)$. Rather, one chooses a tailored measure, call it μ , which should reflect the progress of the algorithm. Nevertheless, in the end we desire an upperbound of the form c^n . Hence, we must assure that there is some constant ℓ such that $\mu \leq \ell n$ during the whole algorithm. Then, a proven upperbound c^μ entails the desired upperbound $c^{\ell n}$.

A simple example. We give an algorithm for DOMINATING SET using less than 2^n steps, see Alg. 2. It branches on vertices by deciding whether they should be in the solution (*active*) or not (*inactive*). Regarding this we call them *active* and *inactive*. A vertex for which this decision has not been made is called *blank*. If a vertex is active, then its neighbors are *dominated*. Let $\text{BLND} = \{v \in V \mid v \text{ is blank \& not dominated}\}$, $\text{INND} = \{v \in V \mid v \text{ is inactive \& not dominated}\}$ and $\text{BLDO} = \{v \in V \mid v \text{ is blank \& dominated}\}$. We now define our measure:

$$\mu = |\text{BLND}| + \omega \cdot (|\text{INND}| + |\text{BLDO}|) \leq n$$

In step 6, we create an EDGE COVER instance $G_{EC} = (V(E_{EC}), E_{EC})$: (1) For all $v \in \text{BLND}$ with $N(v) \cap (\text{BLND} \cup \text{INND}) = \{q\}$, adjoin $e = \{v, q\}$ to E_{EC} and let $\alpha(e) = v$; for all $v \in \text{BLDO}$ with $N(v) \cap (\text{BLND} \cup \text{INND}) = \{x, y\}$, $x \neq y$, put $e = \{x, y\}$ into E_{EC} and (re-)define $\alpha(e) = v$. If C is a minimum edge cover of G_{EC} , set all v active where $v = \alpha(e)$ for some $e \in C$. (2) Set all $v \in \text{BLND} \setminus V(E_{EC})$ with $N(v) \cap (\text{BLND} \cup \text{INND}) = \emptyset$ active. (3) Set all $v \in \text{BLDO}$ such that $|N(v) \cap (\text{BLND} \cup \text{INND})| = 0$ inactive. (4) Set all $v \in \text{BLDO}$ such that $N(v) \cap (\text{BLND} \cup \text{INND}) = \{s\} \not\subseteq V(E_{EC})$ active.

Now we come to the analysis of the branching process in steps 2 and 4. Let $n_{bl} = |N(v) \cap \text{BLND}|$ and $n_{in} = |N(v) \cap \text{INND}|$. If we set v to active in **step 2**, we first reduce μ by one as v vanishes from μ . Then all vertices in $N(v) \cap \text{BLND}$ will be dominated and hence moved to the set BLDO. Thus, μ is reduced by an amount of $n_{bl} \cdot (1 - \omega)$. In the same way the vertices in $N(v) \cap \text{INND}$ are dominated. Therefore, these do not appear in μ anymore. Hence, μ is lowered by $n_{in}\omega$. If we set v inactive we reduce μ by $(1 - \omega)$, as v is moved from BLND to INND. Hence, the branching vector is:

$$(1 + n_{bl}(1 - \omega) + n_{in}\omega, (1 - \omega)) \quad (3)$$

where $n_{bl} + n_{in} \geq 2$ due to step 1. In **step 4**, we have chosen $v \in \text{BLDO}$ for branching. Here, we must consider that in the first and second branch we only get ω as reduction from v (v disappears from μ). But the analysis with respect to $N(v) \cap (\text{BLND} \cup \text{INND})$ remains valid. Thus,

$$(\omega + n_{bl}(1 - \omega) + n_{in}\omega, \omega) \quad (4)$$

is the branching vector with respect to $n_{bl} + n_{in} \geq 3$ due to step 3.

Unfortunately, depending on n_{bl} and n_{in} we have infinite number of branching vectors. But it is only necessary to consider the worst case branches. For (3) these are the ones with $n_{bl} + n_{in} = 2$ and for (4) $n_{bl} + n_{in} = 3$. For any other branching vector, we can find one among those giving a worse upperbound. Thus, we have a finite set of recurrences $R_1(\omega), \dots, R_7(\omega)$ depending on ω . The next task is to choose ω in a way such that the maximum root of the evolving characteristic polynomials is minimum. In this case we easily see that $\omega := 0.5$. Then the worst case branching vector for (3) and (4) is $(2, 0.5)$. Thus, the number of leaves of the search tree evolving from this branching vector can be bounded by $\mathcal{O}^*(1.9052^\mu)$. Thus, our algorithm breaks the 2^n -barrier using a simple measure. So, one might argue that we should use a more elaborated measure to get a better upperbound:

$$\mu' = |\text{BLND}| + \omega_1 \cdot (|\text{INND}|) + \omega_2 \cdot (|\text{BLDO}|)$$

Under μ' , (3) becomes $(1 + n_{bl}(1 - \omega_1) + n_{in}\omega_2, (1 - \omega_2))$; (4) becomes $(\omega_1 + n_{bl}(1 - \omega_1) + n_{in}\omega_2, \omega_1)$. The right choice for the weights turns into a tedious task. In fact, if $\omega_1 = 0.637$ and $\omega_2 = 0.363$, then we get an upperbound $\mathcal{O}^*(1.8899^{\mu'})$. Nevertheless, the current best upperbound is $\mathcal{O}(1.5134^n)$, see [16,46].

Generally, one wants the measure to reflect the progress made by the algorithm best possible. This leads to more and more complicated measures with lots of weights to be chosen. So at a certain point, this task can not be done by hand and is an optimization problem of its own which can only be reasonable solved with the help of a computer. One way of obtaining good weights is to use local search. Starting from initial weights, we examine the direct neighborhood to see if we can find an weight assignment which provides a better upperbound. In practice, this approach works quite well, especially if we use compiled programs. Then an amount of hundreds of characteristic polynomials and several weights can be handled. There is also a formulation as a convex program [20]. For this problem class there are efficient solvers available. An alternative is the approach of Eppstein [6].

5 Correctness of search tree algorithms

In its most primitive form (making complete case distinction at each branch), proving correctness of a search tree algorithm is not a real issue. However, this does become an issue when less trivial branching rules are involved (designed to improve on the running time of the algorithms). This was already noticed in the early days: as said above, correctness was the key issue dealt with in early theory papers on branch-and-bound algorithms. So the question was (and is again): How can we design correct pruning rules that make the search tree shrink as far as possible, without losing the ability to find an optimum solution?

As long as the trivial search tree algorithm is only modified to incorporate certain heuristic priorities with respect to which the branching is performed, no problem with respect to correctness incurs. Such priorities are mere implementations of the inherent nondeterministic selection of a branching item. To further speed up the search, the instance might be modified using reduction rules. As long as these reduction rules are valid for any instance, this is no problem either; however, sometimes such rules are only valid in combination with branching (and a further interplay with the heuristic priorities is possible).

Finally, one might think about transferring the already mentioned ideas about dominance and equivalence of search tree nodes (and the corresponding instances) into the search tree analysis. Here, things become tricky. Recall that upon running a branch-and-bound algorithm, we have dynamically changing information about hitherto best or most promising (partial) solutions at hand. Moreover, we have a certain direction of time in which the underlying static search tree is processed. Both properties give (additional) possibilities to prune search tree nodes, e.g., by not expanding nodes where no improvement over the best solution found so far is to be expected. Nonetheless, it is tempting to stop branching at certain nodes n when it has become clear that other branches would lead to solutions no worse than the ones expected from n . Instead, we would (conceptually) introduce a reference link from n to other nodes in the search tree. However, we must avoid creating cycles in the graph consisting of the search tree together with the conceptual links (viewed as arcs in the graph). An according model named *reference search tree* was defined (and successfully employed) in [44].

6 List of questions and research topics

Is it possible to somehow **use the basic principle of branch-and-bound**, namely the use of a bounding function, **within the run-time analysis of search tree algorithms**? Possibly, the worst case approach is not suitable here. However, are there any possibilities for a reasonable **average-case analysis**? The main mathematical problem and challenge is that, even if we start out with a random instance (in whatever sense), the graph will not be any longer “random” after the first branching operations, since it will have been modified according to some heuristic strategy. So, combinatorial and statistical properties will be

destroyed by branching. However, any progress in this direction would be highly welcome, since the difference between theoretical worst-case analysis (making many search tree approaches seemingly prohibitive) and the actual (fast) run time in practice is far too large and can be best explained by the facts that (1) many heuristics work very nice in practice and (2) a good branching strategy enables to bound the search quite efficiently at a very early stage.

How can **search trees techniques be applied in combination with other algorithmic paradigms** ? This idea has been quite successful in connection with dynamic programming (DP). With DP (when used for solving hard combinatorial problems), often the main problem is the exponential space requirement. In general form, these ideas seem to go back to Morin and Marsten [41], exemplified with the Traveling Salesman Problem (TSP). The simple idea is to trade time and space, more specifically, to use search trees to lower the prohibitive space requirements of DP. More recently, such space saving ideas have been employed to obtain good (not necessarily best) solutions [2], as well as (again) for TSP in [3]. From the early days of research onwards, the connection between search tree algorithms and DP was seen; e.g., in [33]. Karp and Held investigate a kind of reverse question: which discrete decision processes can be solved by DP? One can think to combine search tree algorithms with other paradigms, as, e.g., iterative compression / expansion as known from parameterized algorithms, see [42] for a textbook explanation. However, to the knowledge of the authors, no such research has been carried out yet.

Explore the **measure-and-conquer paradigm within parameterized algorithms**. Only few examples of applying measure-and-conquer to (classical) parameterization are known, see [12] for one such example. However, in those examples, there is always a very close link between the non-parameterized view (say, measured in terms of the number m of edges) and the parameterization (e.g., the number of edges k found in an acyclic subgraph). The finite automata approach of Wahlström [47], as well as the ideas expressed in Sec. 3 that offer the re-interpretation of weights as search-tree reduction might show a way to a broader application of measure-and-conquer to (classical) parameterization. Here, Eppstein's quasiconvex method [6] could be also of interest. Other forms of amortized analysis (using potential functions that are analyzed on each path of the search tree) are reported in [4] and could also find broader applicability.

For specific forms of recurrent (e.g., divide-and-conquer) algorithms, connections between the **run time estimation and fractal geometry** have been shown [5]. How does this setting generalize towards time estimates of search-tree algorithms as obtained by branch-and-bound? Notice that those fractal geometric objects in turn are quite related to finite automata, and there are many connections between finite automata and search trees, see [30,47].

Is there a broader theory behind the idea of **automization of search tree analysis**? Several attempts in the direction of employing computers to do the often nasty and error-prone case-analysis have been reported [7,25,35]. Can such ideas be combined with the measure-and-conquer approach that, in itself, already needs a certain computer assistance?

References

1. D. L. Applegate, R. E. Bixby, V. Chvátal, W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Univ. Press, 2006.
2. M. Bailey, J. Alden, R. L. Smith. Approximate dynamic programming using epsilon-pruning (working paper). TR, University of Michigan, Ann Arbor, 2002.
3. A. Björklund, T. Husfeldt, P. Kaski, M. Koivisto. The Travelling Salesman Problem in bounded degree graphs. *ICALP (1)*, LNCS **5125**, 198–209, 2008.
4. J. Chen, I. A. Kanj, G. Xia. Labeled search trees and amortized analysis: improved upper bounds for NP-hard problems. *ISAAC*, LNCS **2906**, 148–157, 2003.
5. S. Dube. Using fractal geometry for solving divide-and-conquer recurrences (extended abstract). *ISAAC*, LNCS **762**, 191–200, 1993.
6. D. Eppstein. Quasiconvex analysis of backtracking algorithms. *SODA*, 781–790, 2004.
7. S. S. Fedin, A. S. Kulikov. Automated proofs of upper bounds on the running time of splitting algorithms. *IWPEC*, LNCS **3162**, 248–259, 2004.
8. J. Feigenbaum, S. Kannan, M. Y. Vardi, M. Viswanathan. The complexity of problems on graphs represented as OBDDs. *Chicago J. Theor. Comput. Sci.*, 1999.
9. H. Fernau. Two-layer planarization: improving on parameterized algorithmics. *J. Graph Algorithms and Applications*, 9:205–238, 2005.
10. H. Fernau. Parameterized algorithms for HITTING SET: the weighted case. *CIAC*, LNCS **3998**, 332–343, 2006.
11. H. Fernau. A top-down approach to search trees: improved algorithmics for 3-HITTING SET. *Algorithmica*, to appear.
12. H. Fernau, D. Raible. Exact algorithms for maximum acyclic subgraph on a superclass of cubic graphs. *WALCOM*, LNCS **4921**, 144–156, 2008.
13. P. Flajolet, R. Sedgewick. *Analytic Combinatorics*. Cambridge Univ. Press, 2008.
14. F. V. Fomin, S. Gaspers, A. V. Pyatkin, I. Razgon. On the Minimum Feedback Vertex Set Problem: Exact and Enumeration Algorithms. *Algorithmica*, 52.2:293–307, 2008
15. F. Fomin, P. Golovach, D. Kratsch, J. Kratochvil, M. Liedloff. Branch & recharge: Exact algorithms for generalized domination. *WADS*, LNCS **4619**, 508–519, 2007.
16. F. V. Fomin, F. Grandoni, D. Kratsch. Measure and conquer: domination – a case study. *ICALP*, LNCS **3580**, 191–203, 2005.
17. F. V. Fomin, F. Grandoni, D. Kratsch. Solving connected dominating set faster than 2^n . *FSTTCS*, LNCS **4337**, pp. 152–163, 2006.
18. F. V. Fomin, F. Grandoni, D. Kratsch. Measure and conquer: a simple $O(2^{0.288n})$ independent set algorithm. *SODA*, 18–25, 2006.
19. F. V. Fomin, F. Grandoni, A. V. Pyatkin, A. A. Stepanov. Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications. *ACM Trans. Algorithms*, 5:1–17, 2008.
20. S. Gaspers. *Exponential Time Algorithms: Structures, Measures, and Bounds*. PhD thesis, University of Bergen, Norway, 2008.
21. S. Gaspers, M. Liedloff. A Branch-and-Reduce Algorithm for Finding a Minimum Independent Dominating Set in Graphs. *WG*, LNCS **4271**, 78–89, 2006.
22. S. W. Golomb, L. D. Baumert. Backtrack programming. *J. ACM*, 12:516–524, 1965.
23. R. Graham, D. E. Knuth, O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 3. ed., 1989.

24. J. Gramm. *Fixed-Parameter Algorithms for the Consensus Analysis of Genomic Data*. Dissertation, Univ.Tübingen, Germany, 2003.
25. J. Gramm, J. Guo, F. Hüffner, R. Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39:321–347, 2004.
26. S. Hong. *A Linear Programming Approach for the Traveling Salesman Problem*. PhD thesis, The Johns Hopkins University, Baltimore, Maryland, USA, 1972.
27. T. Ibaraki. Theoretical comparison of search strategies in branch-and-bound algorithms. *Intern. J. Computer and Information Sciences*, 5:315–344, 1976.
28. T. Ibaraki. On the computational efficiency of branch-and-bound algorithms. *J. Operations Research Society of Japan*, 26:16–35, 1977.
29. T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *J. ACM*, 24:264–279, 1977.
30. T. Ibaraki. Branch-and-bound procedure and state-space representation of combinatorial optimization problems. *Inf. & Contr.*, 36:1–27, 1978.
31. R. G. Jeroslaw. Trivial integer programs unsolvable by branch-and-bound. *Mathematical Programming*, 6:105–109, 1974.
32. M. Jünger, S. Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Software: Practice and Experience*, 30:1325–1352, 2000.
33. R. M. Karp, M. Held. Finite-state processes and dynamic programming. *SIAM J. Applied Mathematics*, 15:693–718, 1967.
34. D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.
35. A. S. Kulikov. Automated generation of simplification rules for SAT and MAXSAT. *SAT*, LNCS **3569**, 430–436, 2005.
36. O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223:1–72, 1999.
37. O. Kullmann. *Fundamentals of Branching Heuristics*, chapter 7, pages 205–244. In *Handbook of Satisfiability*, IOS Press, 2009.
38. C. J. H. McDiarmid and G. M. A. Provan. An expected-cost analysis of backtracking and non-backtracking algorithms. In *IJCAI 91, Vol. 1*, 172–177, 1991.
39. K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Heidelberg: Springer, 1984.
40. L. G. Mitten. Branch-and-bound methods: general formulation and properties. *Operations Research*, 18:24–34, 1970.
41. T. L. Morin, R. E. Marsten. Branch-and-bound strategies for dynamic programming. *Operations Research*, 24:611–627, 1976.
42. R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Univ. Press, 2006.
43. D. Raible, H. Fernau. A new upper bound for MAX-2-SAT: A graph-theoretic approach. *MFC5*, LNCS **5162**, 551–562, 2008.
44. D. Raible, H. Fernau. Power domination in $O^*(1.7548^n)$ using reference search trees. *ISAAC*, LNCS **5369**, 136–147, 2008.
45. R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32:57–95, 1987.
46. J. M. M. van Rooij, H. L. Bodlaender. Design by measure and conquer, a faster exact algorithm for dominating set. *STACS*, 657–668, 2008.
47. M. Wahlström. *Algorithms, Measures and Upper Bounds for Satisfiability and Related Problems*. PhD thesis, Linköpings universitet, Sweden, 2007.