# Branching Algorithms

Dieter Kratsch

**Abstract**

We provide an introduction to the design and analysis of moderately exponential-time branching algorithms via the study of a collection of such algorithms among them algorithms for Maximum Independent Set, SAT and 3-SAT. The tools for simple running time analysis are presented. The limits of such an analysis including lower bounds for the worst-case running time of some particular branching algorithms are discussed. Some exercices are given.

## Contents

# 1 A First Independent Set Algorithm

We first study a simple branching algorithm for the Maximum Independent Set algorithm. The algorithm uses the standard branching rule in MIS algorithms. "For every vertex $v$ there is a maximum independent set containing $v$, or there is a maximum independent set not containing $v$." Note that when deciding that $v$ is in the independent set then all its neighbors cannot be in it and thus they can be deleted safely. Therefore we may write the standard branching rule of the IS problem, that can be applied to any vertex $v$, as follows:

$$\text{mis}(G) = \max(1 + \text{mis}(G - N[v]), \text{mis}(G - v)).$$

The algorithm applies this rule to any vertex $v$ of degree at most three in the current graph as long as this is possible. The recursion is stopped when the current graph has maximum degree two, i.e. it is a union of cycles and paths. In such a graph a maximum independent set can be computed in polynomial time.

It is not difficult to see that this algorithm is correct and indeed returns the size of a maximum independent set of the input graph. Usually correctness of branching algorithms is not hard to see (if not obvious).

How shall one estimate the overall running time of such a branching algorithm? There is a well-established procedure for such an analysis that is based on using linear recurrences. For our algorithm let $T(n)$ be the largest number of leaves (graphs of maximum degree two) of an input graph on $n$ vertices for which the polynomial time algorithm is called. Then the running time of the algorithm is $O^*(T(n))$. The branching rule implies

$$T(n) \leq T(n-1) + T(n-d(v)-1) \leq T(n-1) + T(n-4).$$

Thus we shall say that our branching rule has branching vector $(1, d(v)+1)$ and the worst case is achieved if $d(v) = 3$. The corresponding branching vector is $(1, 4)$.

It is knwon that all basic solutions of such a linear recurrence are of the form $\alpha^n$ where $\alpha$ is a root of the polynomial

$$x^n = x^{n-1} + x^{n-4}.$$

Since we want to upper bound the running time we are interested in the largest solution of the characteristic polynomial. Fortunately it is known that this is always the unique positive real root of the polynomial. Using some system like Maple, Mathematica, Matlab etc. we obtain that our algorithm has running time $O^*(1.3803^n)$.

This analysis does not seem to take into account what the algorithm is really doing. Somehow with this tool we can analyze branching algorithms without understanding well what happens during an execution. But can this really provide the worst-case running time?

## 2 Fundamental Notions and Tools

We set up the scenario of a typical (moderately exponential time) branching algorithm.

Such an algorithm consists of a collection of *reduction rules* and *branching rules*. There are also halting rules and it needs to be specified which rule to apply on a particular instance. A reduction rule simplifies an instance (without branching). A branching rule recursively calls subproblems of the instance. The correctness of a branching algorithm follows from the correctness of all its reduction and branching rules (which in many cases is easy to see).

Search trees are very useful to illustrate an execution of a branching algorithm and to facilitate the time analysis of a branching algorithm. A *search tree* of an execution of a branching algorithm is obtained as follows: assign the root node of the search to the input of the problem; recursively assign a child to a node for each smaller instance reached by applying a branching rule to the instance of the node. Note that we do not assign a child to a node when an reduction rule is applied. Hence as long as the algorithm applies reduction rules to an instance the instance simplifies but the instance corresponds to the same node of the search tree.

What is the running time of a particular execution of the algorithm on an input instance? To obtain an easy answer, we assume that during its execution the running time of the algorithm corresponding to one node is polynomial. Under this assumption, that is satisfied for all branching algorithms to be presented, the running time of an execution is equal to the number of nodes of the corresponding search tree up to a polynomial factor.

Thus analysing the worst-case running time of a branching algorithm means to determine the maximum number of nodes in a search tree corresponding to the execution of the algorithm on an input of size $n$, where $n$ is e.g. the number of vertices of a graph, the number of variables of a boolean formula.

The time analysis of branching algorithms is based on upper bounding the number of leaves of a search tree of an input of size $n$. Let $T(n)$ be the maximum number of leaves on any search tree of an input of size $n$. Now each branching rule is analyzed separately and finally we use the worst-case time over all branching rules as upper bound of the running time of the algorithm.

Let $b$ be a branching rule of the algorithm to be analyzed. Consider an application of $b$ to any instance of size $n$. Suppose it branches into $r \geq 2$ subproblems of size at most $n-t_1$, $n-t_2$, $\ldots$, $n-t_r$, for all instances of size $n \geq \max\{t_i : i = 1, 2, \ldots r\}$. Then we call $\vec{b} = (t_1, t_2, \ldots t_r)$ the *branching vector* of branching rule $b$. Hence

$$T(n) \leq T(n-t_1) + T(n-t_2) + \cdots + T(n-t_r).$$

It is known that the largest solution of any linear recurrence obtained by a branching algorithm is the unique positive real root of

$$x^n - x^{n-t_1} - x^{n-t_2} - \cdots - x^{n-t_r} = 0.$$

Sometimes this root is called the *branching factor* of branching vector $\vec{b}$.

Having computed the branching factors $\alpha_i$ for every branching vector $b_i$ we simply take the largest base $\alpha_i$ to achieve an upper bound of the running time: $\alpha = \max_i \alpha_i$. Then $T(n) = O^*(\alpha^n)$ and the running time of the branching algorithm is $O^*(\alpha^n)$.

## 3 The Second Independent Set algorithm

The algorithm consists of one branching rule which is based on the fact that if $I$ is a maximal independent set of $G$, then if $v$ is not in $I$, then at least one of the neighbors is in $I$. This is because otherwise $I \cup \{v\}$ would be an independent set, which contradicts the maximality of $I$. Hence the algorithm picks a vertex of minimum degree and for each vertex from its closed neighborhood it recursively computes a maximal independent set of the current graph.

```
1   int mis(G = (V, E)) {
2       if(|V| = 0) return 0;
3       choose a vertex v of minimum degree in G
4           return 1 + max{mis(G − N[y]) : y ∈ N[v]};
5   }
```

To analyze the running time let $G$ be the input graph of a subproblem. Suppose the algorithm branches on a vertex $v$ of minimum degree $d(v)$. Let $y_1, y_2, \ldots y_{d(v)}$ be the neighbors of $v$ in $G$. Thus for solving the subproblem $G$ the algorithm recursively solves the subproblems $G - N[x]$, $G - N[y_1]$, ... ,$G - N[y_{d(v)}]$. Hence we obtain immediately the recurrence

$$T(n) \leq T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(y_i) - 1).$$

Since the algorithm is branching on a vertex of minimum degree, we have: for all $i = 1, 2, \ldots, d(v)$, $d(v) \leq d(v_i)$, $n - d(v_i) - 1 \leq n - d(v) - 1$ and, by the monotonicity of $T$, $T(n - d(v_i) - 1) \leq T(n - d(v) - 1)$. Consequently

$$T(n) \leq T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v) - 1) \leq (d(v) + 1)T(n - d(v) - 1)$$

Taking $s = d(v) + 1$, we establish the recurrence $T(s) \leq s\,T(n-s)$, which has the solution $T(s) = s^{n/s}$. Since this function has its maximum value for integral $s$ when $s = 3$, we obtain $T(n) \leq 3^{n/3}$. Hence the running time of the algorithm is $O^*(3^{n/3})$.

To mention some features of the algorithm. Any set of vertices selected for an independent set in a leave of the search tree is a maximal independent set of the input graph; and each maximal independent set corresponds to at least one leaf of the search tree. Thus the algorithm can be used to enumerate all maximal independent sets of a graph in time $O^*(3^{n/3})$, and hence a graph on $n$ vertices has $O^*(3^{n/3})$ maximal independent sets. This provides a new and simpler proof the well-known combinatorial theorem of Moon and Moser. Since the bound is tight we also obtain that the worst-case running time of the algorithm is $\Theta^*(3^{n/3})$.

# 4 The Third Independent Set algorithm

We discuss various fundamental ideas of branching algorithms for the independent set problem and use them to construct a Third Independent Set algorithm.

The first one is a reduction rule called *domination rule*.

**Lemma 1.** *Let $G = (V, E)$ be a graph, let $v$ and $w$ be adjacent vertices of $G$ such that $N[v] \subseteq N[w]$. Then*

$$\alpha(G) = \alpha(G - w).$$

*Proof.* We have to prove that $G$ has a maximum independent set not containing $w$. Let $I$ be a maximum independent set of $G$ such that $w \in I$. Since $w \in I$ no neighbor of $v$ except $w$ belongs to $I$. Hence $I - w + v$ is an independent set of $G$, and thus a maximum independent set of $G$ not containing $w$. □

Now let us study the branching rules of our algorithm. The *standard branching* has already been discussed:

$$\alpha(G) = \max(\alpha(G - N[v]), \alpha(G - v)).$$

**Lemma 2.** *Let $G = (V, E)$ be a graph and let $v$ be a vertex of $G$. If no maximum independent set of $G$ contains $v$ then every maximum independent set of $G$ contains at least two vertices of $N(v)$.*

*Proof.* Every maximum independent set of $G$ is also a maximum independent set of $G - v$. Suppose there is a maximum independent set $I$ of $G - v$ containing at most one vertex of $N(v)$. If $I$ contains no vertex of $N[v]$ then

$I + v$ is independent and thus $I$ is not a maximum independent set, contradiction. Otherwise, let $I \cap N(v) = \{w\}$. Then $I - w + v$ is an independent set of $G$, and thus there is a maximum independent set of $G$ containing $v$, contradiction. □

Using the above Lemma, standard branching has been refined recently. Let $N^2(v)$ be the set of vertices in distance 2 to $v$ in $G$, i.e. the set of the neighbors of the neighbors of $v$, except $v$ itself. A vertex $w \in N^2(v)$ is called a *mirror* of $v$ if $N(v) \setminus N(w)$ is a clique. Calling $M(v)$ the set of mirrors of $v$ in $G$, the standard branching rule can be refined via mirrors.

**Lemma 3.** *Let $G = (V, E)$ be a graph and $v$ a vertex of $G$. Then*

$$\alpha(G) = \max(\alpha(G - N[v]), \alpha(G - (M(v) + v)).$$

*Proof.* If $G$ has any maximum independent set containing $v$ then $\alpha(G) = \alpha(G - N[v])$ and the lemma is true. Otherwise suppose that no maximum independent set of $G$ contains $v$. Then every maximum independent set of $G$ contains two vertices of $N(v)$. Since $w$ is a mirror the vertex subset $N(v) \setminus N(w)$ is a clique, and thus at least one vertex of every maximum independent set belongs to $N(w)$. Consequently, no maximum independent set contains $w$, and thus $w$ can be safely discarded. □

We call the corresponding rule *mirror branching*.

Lemma 2 also implies the following reduction rule that we call *simplicial rule*.

**Lemma 4.** *Let $G = (V, E)$ be a graph and $v$ be a vertex of $G$ such that $N[v]$ is a clique. Then*

$$\alpha(G) = 1 + \alpha(G - N[v]).$$

*Proof.* If $G$ has a maximum independent set containing $v$ then the lemma is true. Otherwise, by Lemma 2 a maximum independent set must contain two vertices of the clique $N(v)$, which is impossible. □

Sometimes our algorithm uses yet another branching rule. Let $S \subseteq V$ be a (small) separator of the graph $G$, i.e. $G - S$ is disconnected. Then for any maximum independent set $I$ of $G$, $I \cap S$ is an independent set of $G$. Thus we may branch into all possible independent sets of $S$.

**Lemma 5.** *Let $G$ be a graph, let $S$ be a separator of $G$ and let $\mathcal{I}(S)$ be the set of all independent subsets of $S$. Then*

$$\alpha(G) = \max_{A \in \mathcal{I}(S)} |A| \ + \ \alpha(G - (S \cup N[A])).$$

Our algorithm uses the corresponding *separator branching* only under the following circumstances: the separator $S$ is the set $N^2(v)$ and this set is of size at most 2. Thus the branching is done in at most 4 subproblems and for each of it is easy to find out the optimal choice among the vertices of $N[v]$.

The third Independent Set algorithm will be presented and analyzed during the talk.

# 5 Two Algorithms for SAT

First we present the rules of the DPLL algorithm to solve the SAT problem from the early sixties. This branching algorithm has triggered a lot of research in the SAT community and its ideas are used in modern SAT solvers.

Then we study the algorithm of Monien and Speckenmeyer which was the first one with a proven upper bound of $O^*(c^n)$ with $c < 2$ for 3-SAT. Indeed this algorithm solves $k - SAT$ for any fixed $k \geq 3$ in time $O^*({c_k}^n)$ with $c_k < 2$, where $c_k$ depends on $k$.

The algorithm recursively computes CNF formulas obtained by a partial truth assignment of the input $k$-CNF formula, i.e. by fixing the boolean value of some variables and literals, respectively, of $F$. Given any partial truth assignment $t$ of the $k$-CNF formula $F$ the corresponding $k$-CNF formula $F'$ is obtained by removing all clauses containing a true literal, and by removing all false literals. Hence the instance of any subproblem generated by the algorithm is a $k$-CNF formula. The size of a $k$-CNF formula is its number of variables.

We first study the branching rule of the algorithm. Let $F$ be any $k$-CNF formula and let $c = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_t)$ be any clause of $F$. Branching on clause $c$ means to branch into the following $t$ subproblems obtained by fixing the boolean values of some literals as described below:

- $F_1$ : $\ell_1 = \text{true}$
- $F_2$ : $\ell_1 = \text{false}, \ell_2 = \text{true}$
- $F_3$ : $\ell_1 = \text{false}, \ell_2 = \text{false}, \ell_3 = \text{true}$
- $F_t$ : $\ell_1 = \text{false}, \ell_2 = \text{false}, \cdots, \ell_{t-1} = \text{false}, \ell_t = \text{true}$

The branching rule says that $F$ is satisfiable iff at least one $F_i$, $i = 1, 2, \ldots, t$ is satisfiable, and this obviously is correct. Hence recursively solving all subproblem instances $F_i$ we can decide whether $F$ is satisfiable.

Suppose $F$ has $n$ variables. Since the boolean values of $i$ variables of $F$ are fixed to obtain the instance $F_i$, $i = 1, 2, \ldots, t$, the number of (non fixed) variables of $F_i$ is $n - i$. Therefore the branching vector of this rule

is $(1, 2, \ldots, t)$. To obtain the branching factor of $(1, 2, \ldots, t)$ we solve the linear recurrence

$$T(n) \leq T(n-1) + T(n-2) + \cdots + T(n-t)$$

by computing the unique positive real root of

$$x^t = x^{t-1} + x^{t-2} - x^{t-3} - \cdots 1 = 0,$$

which is equivalent to

$$x^{t+1} - 2x^t + 1 = 0.$$

For any clause of size $t$ we denote the branching factor $\beta_t$. Then $\beta_2 \approx 1.6181$, $\beta_3 \approx 1.8393$, $\beta_4 \approx 1.9276$ and $\beta_5 \approx 1.9660$.

We note that on a clause of size 1, there is only one subproblem and thus this is indeed a reduction rule. By adding some simple reduction rules for termination saying that a formula containing an empty clause is unsatisfiable and that the empty formula is satisfiable we would obtain a first branching algorithm consisting essentially of the above branching rule. Of course we may also add the reduction rule saying that if the formula is in 2-CNF then a polynomial time algorithm will be used to decide whether it is satisfiable. The running time of such a simple branching algorithm is $O^*({\beta_k}^n)$ since given a $k$-CNF as input all instances generated by the branching algorithm are $k$-CNF, and thus every clause the algorithm branches on has size $t \leq k$.

Notice that the branching factor $\beta_t$ depends on the size $t$ of the clause $c$ chosen to branch on. Hence it is natural to aim at branching on clauses of as small size as possible. Thus for every CNF formula being an instance of a subproblem the algorithm chooses a clause of minimum size to branch on. Using some nice logic insights one can guarantee that for an input $k$-CNF the algorithm always branches on a clause of size at most $k-1$ (except possibly the very first branching). Such a branching algorithm solves $k$-SAT in time $O^*({\alpha_k}^n)$ where $\alpha_k = \beta_{k-1}$. Hence the algorithm solves 3-SAT in time $O(1.6181^n)$.

## 6 Worst-Case Running Time and Lower Bounds

Lower bounds for the worst-case running time of branching algorithms are of interest since the current tools for the running time analysis of branching algorithms (including Measure & Conquer) seem not strong enough to establish the worst-case running time.

A lower bound of $\Omega^*(c^n)$ to the (unknown) worst-case running time of a particular branching algorithm is established by constructing instances and showing that the algorithm needs running time $\Omega^*(c^n)$ on those instances. Clearly the goal is that lower and upper bound of the worst-case running time of a particular algorithm are close.

**Theorem 1.** *The first Independent Set algorithm has worst case running time $\Theta^*(\alpha^n)$, where $\alpha$ is the branching factor of $(1, 4)$.*

*Proof.* We need to prove the lower bound $\Omega^*(\alpha^n)$. To do this, consider the graph $G_n = (\{1, 2, \ldots, n\}, E)$, where $\{i, j\} \in E \Leftrightarrow |i - j| \leq 3$. For this graph we assume that algorithm will solve ties by always choosing the leftmost remaining vertex to branch on, which always has degree 3. Hence on $G_i$, the algorithm branches into $G_{i-1}$ and $G_{i-4}$. Thus if the search tree generated on $G_n$ has $T(n)$ leaves then $T(n) \leq T(n-1) + T(n-4)$, and thus $T(n) = \Omega(\alpha^n)$. □

Suppose we modify the first Independent Set algorithm such that it branches on a maximum degree vertex. This will not change the upper bound analysis, however the lower bound does not apply anymore. Is this a coincidence or has the modified algorithm really a better worst-case running time?

# 7 Memorization

Memorization in branching algorithms has been introduced by M. Robson. The goal is to speed up the algorithm by storing already computed results in a database to look them up instead of recomputing them many times on different branches of the search tree.

The technique can be used to obtain algorithms with better upper bounds on the running time. Unfortunately the technique leads to algorithms needing exponential space, while the original branching algorithm needs only polynomial space.

# 8 Branch & Recharge

This is a new approach to construct and analyse branching algorithms. The key idea is to explicitely use weights in the algorithm to guarantee that the running time is governed by few recurrences; and thus running time analysis is easy. On the other hand, correctness is no longer obvious and needs a careful analysis of the branching algorithm. A typical operation of such an algorithm is a redistribution of the weights called recharging.

In the algorithm to be presented, every vertex is assigned a weight of 1 at the beginning. A value $\epsilon > 0$ is fixed depending on the problem. Then by a recharging procedure, it is guaranteed that in each branching on any vertex $v$ the overall weight of the input decreases by 1 when not taking vertex $v$ in the solution set $S$, and it decreases by $1 + \epsilon$ when selecting $v$ in $S$. Hence the only branching vector of the algorithm is $(1, 1 + \epsilon)$ and one immediately obtains the running time.

## Exercices

The exercices are ordered from easy ones to research problems.

1. The HAMILTONIAN CIRCUIT problem can be solved in time $O^*(2^n)$ via dynamic programming or inclusion-exclusion. Construct a $O^*(3^{m/3})$ branching algorithm deciding whether a graph has a hamiltonian circuit, where $m$ is the number of edges.
2. Let $G = (V, E)$ be a bicolored graph, i.e. its vertices are either red or blue. Construct and analyze branching algorithms that for input $G, k_1, k_2$ decide whether the bicolored graph $G$ has an independent set $I$ with $k_1$ red and $k_2$ blue vertices. What is the best running time you can establish?
3. Construct a branching algorithm for the 3-COLORING problem, i.e. for given graph $G$ it decides whether $G$ is 3-colorable. The running time should be $O^*(3^{n/3})$ or even $O^*(c^n)$ for some $c < 1.4$.
4. Construct a branching algorithm for the DOMINATING SET problem on graphs of maximum degree 3.
5. Is the following statement true for all graphs $G$. If $w$ is a mirror of $v$ and there is a maximum independent set of $G$ not containing $v$, then there is a maximum independent set containing neither $v$ nor $w$.
6. Modify the first IS algorithm such that it always branches on a maximum degree vertex. Provide a lower bound. What is the worst-case running time of this algorithm?
7. Construct a $O^*(1.49^n)$ branching algorithm to solve 3-SAT.

## References


[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, 2001.

[2] J. Kleinberg, E. Tardos, *Algorithm Design*, Addison-Wesley, 2005.

[3] R.L. Graham, D.E. Knuth, O. Patashnik *Concrete Mathematics*, Addison-Wesley, 1989.

[4] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms* 54:168–204, 2005.

[5] M. Davis, H. Putnam. A computing procedure for quantification theory. *J. ACM* 7:201–215, 1960.

[6] D. Eppstein. The traveling salesman problem for cubic graphs. In *Workshop on Algorithms and Data Structures (WADS)*, pages 307–318, 2003.

[7] F. V. Fomin, S. Gaspers, and A. V. Pyatkin. Finding a minimum feedback vertex set in time $O(1.7548^n)$, in Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC 2006), Springer LNCS vol. 4169, pp. 184–191.

[8] F. V. Fomin, P. Golovach, D. Kratsch, J. Kratochvil and M. Liedloff. Branch and Recharge: Exact algorithms for generalized domination. Proceedings of WADS 2007, Springer, 2007, LNCS 4619, pp. 508–519.

[9] F. V. Fomin, F. Grandoni, D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the EATCS* 87:47–77, 2005.

[10] F. V. Fomin, F. Grandoni, D. Kratsch. Measure and Conquer: A Simple $O(2^{0.288\,n})$ Independent Set Algorithm. Proceedings of SODA 2006, ACM and SIAM, 2006, pp. 508–519.

[11] K. Iwama. Worst-case upper bounds for k-SAT. *Bulletin of the EATCS* 82:61–71, 2004.

[12] K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. Proceedings of SODA 2004, ACM and SIAM, 2004, p. 328.

[13] B. Monien, E. Speckenmeyer. Solving satisfiability in less than $2^n$ steps. Discrete Appl. Math. 10: 287–295, 1985.

[14] I. Razgon. Exact computation of maximum induced forest. *Proceedings of the 10th Scandinavian Workshop on Algorithm Theory (SWAT 2006).* Springer LNCS vol. 4059, p. 160-171.

[15] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms* 7(3):425–440, 1986.

[16] R. Tarjan and A. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[17] G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. *Combinatorial Optimization – Eureka, You Shrink*, Springer LNCS vol. 2570, 2003, pp. 185–207.

[18] G. J. Woeginger. Space and time complexity of exact algorithms: Some open problems. Proceedings of the *1st International Workshop on Parameterized and Exact Computation (IWPEC 2004)*, Springer LNCS vol. 3162, 2004, pp. 281–290.

[19] G. J. Woeginger. Open problems around exact algorithms. *Discrete Applied Mathematics*, 156:397–405, 2008.