Branching

Teppo Niinimäki

Helsinki October 14, 2011

Seminar: Exact Exponential Algorithms

UNIVERSITY OF HELSINKI
Department of Computer Science

For a large number of important computational problems no polynomial-time algorithm is known. The family of NP-complete problems is the best known family of such problems. Every combinatorial problem can be solved by using a brute force search that explicitly tests all possible solutions, but often better alternatives are available.

Branching is one of the basic techniques for designing faster exponential algorithms for such problems. It is based on dividing the problem recursively into smaller subproblems. This strategy has led to significant improvements over trivial brute force algorithms in computational complexity for problems such as k-Satisfiability or Maximum Independent Set. The ideal behind branching is simple and intuitive and has been invented multiple times under alternative names such as branch and reduce, splitting, backtracking or pruning search tree.

The rest of this paper is organized as follows: In section 1 the general structure and principles of branching algorithms are described. In section 2 a branching algorithm for the k-Satisfiability problem is presented and analysed. Finally in section 3 we look very briefly into how branching can be used to build fast algorithms for the Maximum Independent Set problem. The material in this paper is based on the Chapter 2 of the book Exact Exponential Algorithms by Fomin and Kratsch (2010).

1 General Framework

Let n be the size of the problem such as the number of nodes in a graph. Note, that n does not need to be the size of the actual input, but instead it can be chosen quite freely. In general, any branching algorithm consists of two kinds of rules: *Branching rules* convert the problem to two or more smaller problems which are then solved recursively, and *reduction rules* simplify the problem or halt the algorithm. Recursive application of branching rules results in a tree structure which is called *search tree*. The root of the search tree represents the original problem and other nodes represent smaller instances obtained by the use of branching rules. The time spend in one node is assumed to be polynomial in the size of the problem, which also includes the time spent in applying possible reduction rules. Thus the running time of the algorithm is a polynomial times the number of nodes of the search tree, which is normally $\mathcal{O}^*(\alpha^n)$ for some constant $\alpha > 1$. Since we are not interested in polynomial factors, we only need to concentrate on branching rules when analysing the running time and reduction rules can generally be ignored.

	1	2	3	4	5	6
1	2.0000	1.6181	1.4656	1.3803	1.3248	1.2852
2	1.6181	1.4143	1.3248	1.2721	1.2366	1.2107
3	1.4656	1.3248	1.2560	1.2208	1.1939	1.1740
4	1.3803	1.2721	1.2208	1.1893	1.1674	1.1510
5	1.3248	1.2366	1.1939	1.1674	1.1487	1.1348
6	1.2852	1.2107	1.1740	1.1510	1.1348	1.1225

Table 1: Branching factors for binary branching vectors, rounded up.

As no general method for determining the worst-case running time of branching algorithm is known, an upper bound for α is normally searched. Since the number of leaves in any search tree is always at least half of the number of all nodes we can concentrate on finding an upper bound for the number of leafs.

Let b be a branching rule that divides a problem of size n into $r \geq 2$ smaller problems of sizes $n - t_1, n - t_2, \ldots, n - t_r$, where $t_i > 0$ for all $i = 1, 2, \ldots, r$. Then $\mathbf{b} = (t_1, t_2, \ldots, t_r)$ is called the *branching vector* of branching rule b. Let $T(n)$ be the maximum number of leaves in a search tree of an input problem of size n. We get the following linear recurrence:

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

It is known that the solution of this linear recurrence is of the form $T(n) = \alpha^n$ where α can be obtained by finding the unique positive real root of the corresponding equation $x^n - x^{n-t_1} - x^{n-t_2} - \cdots - x^{n-t_r} = 0$. This constant, denoted by $\alpha = \tau(t_1, t_2, \ldots, t_r)$, is called the *branching factor* of branching vector $\mathbf{b}$. As a result the running time of an algorithm which uses only branching rule b is $\mathcal{O}^*(\alpha^n)$.

Branching vectors with two elements are called *binary*. Table 1 contains the branching factors for some common binary branching vectors.

There are some important properties concerning branching factors. First, it is easy to see that the order of elements in branching vector does not affect the branching factor. In addition the following two properties hold:

- If $t_1 > t'_1$, then $\tau(t_1, t_2, \ldots, t_r) < \tau(t'_1, t_2, \ldots, t_r)$. In this case it is said that the latter branching vector dominates the former.
- If $0 < a < b$, then $\tau(a+\epsilon, b-\epsilon) < \tau(a, b)$ for all $0 < \epsilon < (b-a)/2$. So intuitively it is beneficial to make the branching vector as balanced as possible.

Above we saw how to analyse an algorithm with only one branching rule. Typically a branching algorithm consists of multiple branching rules and in each node of the search tree one of these rules is selected to be applied based on various criteria. In the case of multiple branching rules the upper bound for the running time is $\mathcal{O}^*(\alpha^n)$ where α is the maximum of the branching factors of individual branching rules.

While in a general case the rule with the highest branching factor determines the worst case running time, it is not unusual that the analysis can be tightened. If it is known that a branching rule with a high branching factor is always followed by another rule with a low branching factor, an analysis technique called the *addition of branching vectors* can be used. For example, suppose that after branching with branching vector (i, j) in the left branch (corresponding to i) a branching rule with a branching vector (k, l) is always immediately applied. In this case we can treat these two branching rules as one with a combined branching vector $(i + k, i + l, j)$.

2 k-Satisfiability

This section describes a classical branching algorithm by Monien and Speckenmeyer for the k-Satisfiability problem. The algorithm is best known for the fact that it solves the 3-Satisfiability problem in time $\mathcal{O}^*(1.6181^n)$.

Let $\{x_1, x_2, \ldots, x_n\}$ be a set Boolean variables. A clause $c = (l_1 \vee l_2 \vee \cdots \vee l_t)$ is a disjunction of literals where each literal l is a Boolean variable x or its negation $\bar{x}$. A Boolean formula $F = (c_1 \wedge c_2 \wedge \cdots \wedge c_r)$ which is a conjunction of clauses is said to be in *conjunctive normal form* (CNF). If every cause contains at most k literals the formula is a *k-CNF formula*. We represent a clause c by the set of its literals $\{l_1, l_2, \ldots, l_t\}$ and a formula F by the set of its clauses $\{c_1, c_2, \ldots, c_m\}$.

F is said to be satisfiable if there is a truth assignment of variables such that F evaluates to true. For a CNF formula this means that every clause must contain at least one true literal. An empty formula is always true and thus satisfiable. On the other hand, empty clause is always false and makes the formula unsatisfiable. The problem of deciding whether a given formula is satisfiable is called the Satisfiability problem (`SAT`). If the input is in k-CNF form the problem is called k-Satisfiability (k-`SAT`). From now on we only handle k-`SAT` and thus any formula F is assumed to be in k-CNF form.

Let now k be fixed and n be the number of variable. The size of the formula is

denoted by n. Because there are at most $2n$ different literals, the number of different clauses in F is at most $m \leq \sum_{i=0}^{k} \binom{2n}{i}$ and is thus upper bounded by a polynomial. It is known that 2-SAT is solvable in linear time but for $k \geq 3$ the k-SAT problem is NP-complete. Any k-SAT can be solved by a trivial brute force algorithm that separately examines every possible truth assignment. Since there are 2^n different truth assignments and evaluating for one truth assignment takes polynomial number of steps, the brute force algorithm requires $\mathcal{O}^*(2^n)$ time.

We now describe a branching algorithm, that solves the k-SAT problem by recursively making partial truth assignments. In a partial truth assignment the truth values of some of the variables are fixed. When a partial truth assignment is applied to formula F, a new simplified formula F' is obtained by removing all clauses that contain a true literal and removing all false literals from the remaining clauses.

The algorithm now works as follows: Let F be the input formula. If F is empty, it is satisfiable and the algorithm returns true. If F contains an empty clause, the algorithm returns false. Otherwise, one clause is selected and a branching rule is applied. Let $(l_1 \vee l_2 \vee \cdots \vee l_q)$ be the selected clause. The algorithm branches into q simplified cases $F_1, F_2, \ldots, F_q$ corresponding the following partial truth assignment:

$$
\begin{aligned}
&F_1 : \ l_1 = \text{true} \\
&F_2 : \ l_1 = \text{false}, \ l_2 = \text{true} \\
&\vdots \\
&F_q : \ l_1 = \text{false}, \ l_2 = \text{false}, \ \ldots, \ l_{q-1} = \text{false}, \ l_q = \text{true}
\end{aligned}
$$

It is obvious that F is satisfiable if and only if at least one of the resulting simplified formulas is satisfiable. If the selected clause contains only one literal this is actually a reduction rule. Otherwise, we get a branching vector $(1, 2, \ldots, q)$.

For the given branching rules the branching factors $\beta_q = \tau(1, 2, \ldots, q)$ for first few clause sizes are $\beta_2 < 1.6181$, $\beta_3 < 1.8393$ and $\beta_4 < 1.9276$ (rounded up). Because for a k-CNF formula the results of the branching and reduction rules are also k-CNF formulas, the branching factor is always at most β_k and the running time of the algorithm is $\mathcal{O}^*(\beta_k^n)$. For the 3-SAT problem this is $\mathcal{O}^*(1.8393^n)$.

There is relatively simple way to improve the above algorithm. It is based on the observation, that if we were able to guarantee that the branching always (except possibly the very first branching) happens on a clause of size at most $k-1$, the worst case branching factor would be β_{k-1}.

Let again F be the input formula. In partial truth assignment all clauses contain-

ing fixed true literals and the remaining fixed false literals are removed from the resulting formula F'. If any of the remaining clauses contains a fixed literal then the literal must be false and the size of that clause is dropped by at least one and thus on the next branching step we can choose a clause of size at most $k-1$ from F'. Unfortunately this is not always the case. However, if none of the remaining clauses contains a fixed literal, then we know how to satisfy the removed clauses independently of the remaining clauses and thus F is satisfiable if and only if F' is satisfiable. This latter kind of partial truth assignment for F is called *autark*.

Based on the above observations we make the following modification the the previous algorithm. Instead of any clause, a clause of minimum size is always selected for branching. Before actually performing the branching the algorithm checks whether any of the partial truth assignments is an autark. If this is the case, no branching is done and instead a reduction to the corresponding simplified formula F' is performed. This is a reduction rule. If none of the partial truth assignments is autark, the algorithm branches normally.

In the case of normal branching every subformula $F_1, F_2, \ldots, F_q$ contains a clause of size at most $k-1$ and thus the branching factors of the subsequent branchings are at most β_{k-1}. On the other hand, after the autark reduction it is possible that the resulting formula F' contains only clauses of size k leading to branching vector $(1, 2, \ldots, k)$. But because the size of the formula is decreased by at least one in the reduction step, the branching vector actually becomes at least $(2, 3, \ldots, k+1)$ if we consider F as an input instead of F'. It can be shown that $\tau(2, 3 \ldots, k+1) < \tau(1, 2, \ldots, k-1)$ and thus $\beta_{k-1} = \tau(1, 2, \ldots, k-1)$ is the dominating branching factor. We conclude that the modified algorithm solves k-SAT in time $\mathcal{O}^*(\beta_{k-1}^n)$. In particular, for 3-SAT the running time is $\mathcal{O}^*(1.6181^n)$.

3 Maximum Independent Set

In this section we make a short look on how to use branching to develop a reasonably fast exponential algorithm for the Maximum Independent Set problem. We do not describe the whole algorithm in detail, but just the general reduction and branching rules behind the algorithm.

Let $G = (V, E)$ be an undirected graph. A set $I \subset V$ of vertices is an independent set if none of the pairs of vertices from I is adjacent in G. We denote by $\alpha(G)$ the maximum size of an independent set of G. In the Maximum Independent Set

problem (MIS) the task is to find an independent set of maximum size.

We use the following notations: Let $v \in V$ be a vertex. The *neighborhood* of v is $N(v) = \{u \in V : (v, u) \in E\}$ and the *open neighborhood* of v is $N[v] = N(v) \cup \{v\}$. Respectively, the set of vertices at distance 2 from v, that is, the set the set of neighbors of the neighbors of v, is denoted by $N^2(v)$. For a subset $S \subseteq V$ of vertices $G[S]$ is a subgraph of G that has S as a vertex set and those edges from E that have both endpoints in S. The subgraph $G[V \setminus S]$ is denoted by $G \setminus S$ and $G \setminus \{v\}$ for a vertex v can be shortened to $G \setminus v$. The closed neigborhood of S is denoted by $N[S] = \bigcup_{v \in S} N[v]$.

Let G be an input graph. For the MIS problem the following reduction rules can be applied:

1. (*domination rule*) If v and w are adjacent vertices and $N[v] \subseteq N[w]$, then
$$\alpha(G) = \alpha(G \setminus w).$$
2. (*simplicial rule*) If v is a vertex and $N[v]$ is a clique, then
$$\alpha(G) = 1 + \alpha(G \setminus N[v]).$$

In addition to the two reduction rules above the following branching rules can be derived:

3. (*standard branching*) If v is a vertex, then
$$\alpha(G) = \max(1 + \alpha(G \setminus N[v]), \alpha(G \setminus v)).$$
4. (*mirror branching*) If v is a vertex, then
$$\alpha(G) = \max(1 + \alpha(G \setminus N[v]), \alpha(G \setminus (M(v) \cup \{v\})),$$
where $M(v) = \{w \in N^2(v) : N(v) \setminus N(w)$ is a clique$\}$.
5. (*separator branching*) If $S \subseteq V$ and $G \setminus S$ is disconnected, then
$$\alpha(G) = \max_{A \in \mathcal{I}(S)} |A| + \alpha(G \setminus (S \cup N[A])),$$
where $\mathcal{I}(S)$ contains the subsets of S which are independent sets of G.
6. If G is disconnected and $C \subseteq V$ is a connected component of G, then
$$\alpha(G) = \alpha(G[C]) + \alpha(G \setminus C).$$

Using the above reduction and branching rules it is possible to construct a relatively fast exponential algorithm for the Maximum Independent Set problem. The basic idea is to select on each step a rule from above based on the minimum and maximum degree of the input graph. Such algorithm with running time $\mathcal{O}^*(1.2786^n)$ as well as the proofs of the above rules are described in more detail in (Fomin and Kratsch, 2010).

References

Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*, chapter 2. Branching, pages 13–30. Springer, 2010.