Measuring Search Trees

Christian Bessière\textsuperscript{1}, Bruno Zanuttini\textsuperscript{2}, and Cèsar Fernández\textsuperscript{3}

\textsuperscript{1} LIRMM-CNRS, Montpellier, France
\textsuperscript{2} GREYC, Caen, France
\textsuperscript{3} Univ. de Lleida, Lleida, Spain

Abstract. The SAT and CSP communities make a great use of search effort comparisons to assess the validity of an algorithm or a heuristic. There exist different ways of counting the size of a search tree developed by a procedure. However, they are seldomly defined, and their similarities/differences are not clear. This note aims at clarifying this issue. We formally define (some of) the existing measures and characterize the properties that we can expect from them. We show some weaknesses of the existing measures with respect to those properties. We propose the number of wrong decisions as a new way of measuring the search effort that is more accurate to effectively represent the effort an algorithm has devoted to the search. We show that this new way of measuring the search space behaves well with respect to the desired properties.

1 Introduction

In the satisfiability and constraint communities, we often need to compare the relative performance of different algorithms. There are different measures that can be used, and that have advantages and/or drawbacks.

In the constraint community, the number of times a constraint has been checked as satisfied or not (constraint check) has been used for a long time. It is considered as the basic operation all algorithms do. It has also some advantages such as being proportional to the quantity of information retrieved from the constraint network during the solving process. But it has shortcomings which are evident as soon as we use complex algorithms that perform other types of computation than just checking constraints. (AC-4 was an obvious example of an algorithm storing once and for all the constraint checks, and then working only with its data structure.)

Another measure is the cpu time. It has the advantage of taking into account all the parts of the effort devoted to the solving process, but is affected by the quality of the implementation. This issue is particularly accurate when we want to compare the performance of an algorithm implemented on site with a "standard" one available on the web or in a market product. The two versions are not necessarily implemented with the same skills.

The last measure, which is widely used, both in CSP and SAT solvers (at least in those using backtrack search), is the size of the search tree explored during the solving process. This size is often evaluated as the number of backtracks.
But, if we check what is called “number of backtracks” by different people, inside different solvers, or in different papers, it appears that this concept is rather fuzzy, and different ways of measuring the size of the search tree can be reported under this name. The goal of this note is to clarify our picture on these measures. We present them and provide a formal definition. We propose a new measure, the number of wrong decisions, that is more consistent with what is effectively done during the solving process of an NP-complete problem. We characterize properties that we can expect from a search measure. We show that all measures but ‘wrong decisions’ fail on one of the properties.

2 Search Trees

We give some definitions that will permit to handle more formally the notion of a search tree. These definitions are based on a standard CSP understanding, but all the definitions hold for SAT, or any similar type of search.

A search tree $T$ is composed of nodes, arcs, and an ordering $<_c$ on the outgoing arcs of each node. A node $u$ represents an ordered partial instantiation $I(u) = (x_1 = v_1, \ldots, x_n = v_n)$. A search tree is rooted at the particular node $u_0$ with $I(u_0) = \emptyset$. There is an arc from a node $u$ to a node $u_1$ if $I(u_1) = I(u) \cup \{x = v\}$, $x$ and $v$ being a variable and one of its values. ($\cup$ denotes concatenation.) $u_1$ is called a child of $u$, and $u$ the parent of $u_1$. For every node $u$, $T(u)$ denotes the subtree of $T$ rooted at $u$. The set $ch(u, T)$ of the children of $u$ in $T$ is totally ordered by $<_c$. The search tree $T_{\text{full}}^A$ of a backtrack search algorithm $A$ solving a particular problem $P$ is the search tree such that there is a one-to-one mapping between nodes in the tree and instantiations visited by $A$ until it reached a solution or proved inconsistency of $P$. Given two nodes $u_1$ and $u_2$ children of a node $u \in T_{\text{full}}^A$, $u_1 <_c u_2$ iff $I(u_1)$ has been visited by $A$ before $I(u_2)$. The complete search tree $\overline{T}_{\text{full}}^A$ of $A$ on $P$ is the search tree developed by $A$ if $A$ was not stopped after a solution is found, but continued until the end of the enumeration.

3 Search Cost

We define several ways of counting the search cost of an algorithm $A$ on a particular problem $P$. All but the last one have already extensively been used in papers comparing search algorithms:

1. number of nodes,
2. number of backtracks,
3. number of decisions,
4. number of wrong decisions.

We now define these four concepts, and we will illustrate their differences on an example (Fig. 2). For all of them we also discuss where a counter should be placed in an algorithm for computing the measure. For that purpose, we give
function BT_search(int level) /* |I|= level */
1. if preprocessing(level) then
2. | if termination condition then return 1
3. | select a variable $x_i$
4. | for each value $v_i$ in dom($x_i$) do
5. | | assign $v_i$ to $x_i$
6. | | if BT_search(level+1) then return 1
7. undo(level)
8. return 0

Fig. 1. Backtrack scheme

the general scheme of a backtrack-like algorithm on Fig. 1. The procedure is initially called with level=0. The function preprocessingO can contain any simplification algorithm. It usually prunes the domains with respect to a given local consistency: unit propagation in DPLL for CNF formulas, bound consistency in constraint solvers involving numerical constraints, or some form of arc consistency in a CSP search algorithm such as forward checking [HE80] or MAC [SF94]. In the basic backtrack algorithm [GB65], it just checks whether the new assignment together with the current one satisfies all the relevant constraints.

In all cases, we assume that preprocessingO fails if the domain of a variable is empty.

Finally, we assume that the recursive call on Line 6 of Fig. 1 may modify the domain of $x_i$. Think indeed of a backjump algorithm for instance, where the preprocessing on the $i$th level could empty the domain of the $i$-th variable because the culprit assignment is that to the $i$-2th one.

Number of nodes. The simplest technique is to count the number of nodes visited by a procedure. The definition is obvious.

Definition 1 ($\#nodes$). *The number of nodes of the resolution of a problem $P$ by an algorithm $A$ is simply the number of nodes in $T^A_P$.*

Counting the number of nodes visited by an algorithm on a given problem is obvious as well: just increment the counter by 1 each time the function BT_search is entered. This means adding the following line to Fig. 1:

0. $\#nodes++$

A weakness of this measure is that it doesn't discriminate between the effort devoted to traversing the tree “top-down” and “left-right”. In other words, visiting $n$ failing nodes, or going deterministically from the root to a leaf without any wrong choice will add the same $n$ nodes to the count ($n$ is the number of variables), while in the second case not a single mistake was done.
**Number of backtracks.** This is the best known measure. It counts the number of times a procedure goes back from a variable \(x_i\) to its predecessor \(x_{i-1}\), after having proved that none of the extensions of \(I(x_i)\) can be extended to a solution. In terms of the search tree, it counts the number of times the search goes up in the tree from a node \(u\) to its predecessor after having exhausted the last child of \(u\). More formally:

**Definition 2 (**\(\#\text{backtracks}\)**). The number of backtracks of the resolution of a problem \(P\) by an algorithm \(A\) is equal to \(|\{u \in T^A_P | ch(u,T^A_P) \neq \emptyset \land \neg P \land I(u) \rightarrow \bot\}\)|.

The condition for a node \(u\) to be counted as a backtrack node is that it has some children which have been visited (first component of the condition), and that none of them was successful (second component).

Counting the backtracks of \(P\) on \(A\) is quite straightforward: increment the counter by 1 each time the function \(BT\_search\) fails after having tried to extend the current instantiation. This means adding the following line on Fig. 1:

6bis. | \#backtracks++

A major drawback of this way of counting the search effort is that it doesn’t take into account the number of values tried for a variable. In other words, a given node \(u\) can have 2 or 100 children, we will count them both as one backtrack, if the last child fails.

**Number of decisions.** This measure is used in several modern SAT solvers available on the web [MMZ+01,EY02]. It counts the number of times a choice had to be made during the search. This prevents us from counting instantiations that were obvious, as when a domain is a singleton. (As opposed to \(#\text{nodes}\).)

**Definition 3 (**\(\#\text{decisions}\)**). The number of decisions of the resolution of a problem \(P\) by an algorithm \(A\) is

\[
\sum_{u \in T^A_P} \max(0, |ch(u,T^A_P)| - 1) + \{|u \in T^A_P | \emptyset \subset ch(u,T^A_P) \subset \overline{ch(u,T^A_P)}\} \}

The formula identifies the number of decisions taken in each node \(u\) to its number of children minus 1 (first component), and adds one for each node such that the last explored child was not the last possible one—it was itself a decision (second component).

In order to count the number of decisions of an algorithm on a problem, we simply count how many times a value is selected while it was not the only alternative for \(x_i\):

4bis. | | if \(v_i\) is not the only remaining value for \(x_i\) then
| | | \#decisions++

Whereas this measure has advantages over \(#\text{nodes}\), it still has the same drawback as \(#\text{nodes}\), namely, counting choices that will be successful. In other words, we can obtain a score of \(n\) for a backtrack-free search in which all the choices were directly the good ones.
Number of wrong decisions. This measure tries to overcome all the weaknesses pointed out so far. As opposed to \#nodes or \#decisions, it reports “0” if a search was successful on all its choices. But as opposed to \#backtracks, it takes into account the number of children a given node has. It counts the number of times a choice had to be made during the search, except if this choice finally leads to a solution.

Definition 4 (\#wrong decisions). The number of wrong decisions of the resolution of a problem \( P \) by an algorithm \( A \) is

\[
\sum_{u \in T^A_P} \max(0, |\h(u, T^A_P)| - 1)
\]

This definition is expressed in terms of the search the algorithm \( A \) performs on the problem \( P \), but we can also give an alternative one.

Property 1 For any algorithm \( A \) and problem \( P \), the number of leaves in \( T^A_P \) equals \#wrong-decisions\((A,P)\) + 1.

Proof. This is easily seen by induction on \( T^A_P \). Indeed, if \( T^A_P \) has only one node, and thus one leaf the number of wrong decisions is 0. Now if the root of \( T^A_P \) has \( k \) children, write \#leaves\(_i\) for the number of leaves in its \( i \)th child and \#wd\(_i\) for the number of wrong decisions in the subproblem defined by the \( i \)th child. Then the number of leaves in \( T^A_P \) is \( \sum_{i=1}^{k} \text{#leaves}_i \), i.e., \( \sum_{i=1}^{k} (\text{#wd}_i + 1) \) by the induction hypothesis. This is \( k + \sum_{i=1}^{k} \text{#wd}_i = 1 + ((k-1) + \sum_{i=1}^{k} \text{#wd}_i) \), which is exactly the number of wrong decisions of \( A \) on \( P \) plus 1.

The intuition behind this equivalence is that the number of wrong decisions in a search counts exactly the partial assignments that are explored while they cannot be extended into a solution, i.e., the leaves of the search tree, except for the last one that is either a solution or the last element of a proof that the problem is inconsistent.

As for counting the wrong decisions of an algorithm \( A \) on a problem \( P \), increment a counter by 1 each time \( A \) undoes the modifications coming from the selection of a value \( v_i \) if this was not the last possibility for \( x_i \):

6bis. | | if there are still some values in dom(\( x_i \)) then
      | | #wrong-decisions++

The four measures above are illustrated on a search tree on Fig. 2.

4 Properties of measures

We now formally define two properties that we should expect from a measure of the search cost, and evaluate those measures defined in Section 3 with respect to them. These two properties are the following:

1. strict monotonicity,
2. stability under binarization of the search tree.
4.1 Strict monotonicity

The first property that we can expect from a measure is very natural: we want that the measure of a search tree $T_P^A$ is more than that of a search tree $T_P^{A'}$ that is included in $T_P^A$. Inclusion here is intended in the following sense: $T_P^{A'}$ is said to be included in $T_P^A$ if every node in $T_P^{A'}$ is in $T_P^A$ and there is at least one node $u$ that has $k \geq 1$ children in $T_P^{A'}$ and more than $k$ in $T_P^A$. The latter condition allows to consider that two search trees are equivalent if one is obtained from the other only by extending some of its branches; indeed, the search space explored in this case is essentially the same.

**Definition 5 (strict monotonicity).** A measure of search cost $\mu$ is said to be strictly monotonic if for any two search trees $T_P^A, T_P^{A'}$ such that $T_P^A$ is included in $T_P^{A'}$, $\mu(T_P^A) < \mu(T_P^{A'})$ holds.

**Property 2** Measures $\text{#nodes}$ and $\text{#wrong decisions}$ are strictly monotonic, while $\text{#backtracks}$ and $\text{#decisions}$ are not.

*Proof.* As for $\text{#backtracks}$ and $\text{#decisions}$, a counter-example is given on Fig. 3. The tree on the left is indeed included in that on the right, but it is easily seen that $\text{#decisions}=3$ and $\text{#backtracks}=1$ for both (assuming the problem is consistent). As for $\text{#nodes}$, it is obviously strictly monotonic, and finally $\text{#wrong decisions}$ is as well because if $T_P^A$ is included in $T_P^{A'}$ then $T_P^{A'}$ has at least one more leaf than $T_P^A$. \hfill $\square$

4.2 Binary trees

More and more CSP techniques, such as the well-known MAC procedure [SF94], use refutation at each node of the tree. Thus, these procedures develop a binary tree. The left branch is a choice of a value, and the right branch is the refutation of this value, except if there remains a single value in the domain in which
case the right branch instantiates that remaining value. (See [Mit03,KB03] for a comparison of regular and binary CSP tree search.) We give a general scheme for binary tree search on Fig. 4.

```
function Binary_BT_search(int level) /* |I|= level */
1. if preprocessing(level) then
2.  | if termination condition then return 1
3.  | select a variable xi with |dom(xi)|>1
4.  | select a value vi in dom(xi) and assign it to xi
5.  | if Binary_BT_search(level+1) then return 1
6.  | dom(xi) := dom(xi)-{vi}
7.  | if Binary_BT_search(level+1) return 1
8.  undo(level)
9.  return 0
```

**Fig. 4.** Binary backtrack scheme

Binary tree search is more general than non-binary since it is possible to change not only the value, but also the variable after each failure (lines 3 and 4 on Fig. 4). In addition, any non-binary search tree $T^b_B$ has a binarized counterpart, noted $B^b_B$, which corresponds to the binary search in which the selection of variables and values are done in exactly the same ordering as in the non-binary one, and function `preprocessing()` in line 1 does nothing when its call does not follow an instantiation (in line 7).

The binarized counterpart of the search tree of Fig. 2 is presented on Fig. 5.

The definitions given in Section 3 still hold for binary search trees, with counters added to the scheme on Fig. 4 in the following way:

1. for number of nodes,
   
   ```
   #nodes++
   ```

2. for number of backtracks,
Fig. 5. The binarized counterpart of the search tree in Fig. 2.

7bis. | #backtracks++
3. for number of decisions,

3bis. | #decisions++
4. for number of wrong decisions.

5bis. | #wrong-decisions++

It is natural to expect from a measure of the search cost that it allows us to compare non-binary backtrack algorithms and binarized ones. In particular, we expect from a measure that the cost of a non-binary search tree is the same as that of its binarized counterpart, since the search space explored is essentially the same. The following definition formalizes the stability of a measure:

**Definition 6 (stability under binarization).** A measure of search cost $\mu$ is said to be stable under binarization if for any two search trees $T^A_P, B^A_P$ such that $B^A_P$ is the binarization of $T^A_P$, $\mu(T^A_P) = \mu(B^A_P)$ holds.

**Property 3** Measures #decisions and #wrong decisions are stable under binarization, while #nodes and #backtracks are not.

**Proof.** As for #nodes and #backtracks, the tree on Fig. 2 and its binarization on Fig. 5 give a counter-example for both. As for #wrong decisions, it is easily seen that the number of leaves in the binarized tree is the same as in the original one, and thus Proposition 1 concludes.

Thus we are left with #decisions; we show the result by induction on $T^A_P$. Denote by $B^A_P$ the binarization of $T^A_P$, and first remark that if $T^A_P$ has only one
node then $B^A_p = T^A_p$. Now assume the root of $T^A_p$ has $k \geq 1$ children, and write $\# \text{dec}_i$ for the number of decisions in the subproblem defined by its $i$th child. By definition of binarized searches $B^A_p$ can take one of two forms, depending on whether there are more than $k$ values in the domain of the first variable. These two forms are depicted on Figure 6: that on the middle corresponds to the case where there are more than $k$ values, and that on the right to that where there are exactly $k$ of them.

As for the general tree, by definition the number of decisions of $A$ on $P$ is $k + \sum_{i=1}^{k} \# \text{dec}_i$ in the first case and $k - 1 + \sum_{i=1}^{k} \# \text{dec}_i$ in the second case. Now in the first case, each node $u_i$, $i < k$ in $B^A_p$ adds 1 to the number of decisions since it has two children and $|ch(u_i, B^A_p)| = |ch(u_i, \overline{B^A_p})|$, and node $u_k$ adds 1 as well since it has only one child but $|ch(u_k, B^A_p)| \neq |ch(u_k, \overline{B^A_p})|$, thus the number of decisions in the binarized search is $k + \sum_{i=1}^{k} \# \text{dec}_i$ by the induction hypothesis, that is, the number of decisions in the general search. Finally, in the second case each node $u_i$, $1 \leq i \leq k - 1$ adds 1 to the number of decisions since it has two children and $|ch(u_i, B^A_p)| = |ch(u_i, \overline{B^A_p})|$, thus the total number of decisions is $k - 1 + \sum_{i=1}^{k} \# \text{dec}_i$, which again equals the number of decisions in the general search.

5 Summary and Conclusion

We have contributed to a clarification of the different ways of measuring the size of a search tree developed by a backtrack procedure. We have formally defined the existing measures and we have characterized some properties that can be expected from such measures. We have shown some weaknesses of the existing measures with respect to those properties. We have proposed the number of wrong decisions as a new way of measuring the search effort. This new measure behaves well with respect to the desired properties.
References


