Int. J. Appl. Math. Comput. Sci., 2014, Vol. 24, No. 4, 901–916
DOI: 10.2478/amcs-2014-0066

ACCELERATING BACKTRACK SEARCH WITH A BEST–FIRST–SEARCH STRATEGY

ZOLTÁN ÁDÁM MANN, TAMÁS SZÉP

Department of Computer Science and Information Theory
Budapest University of Technology and Economics, Magyar tudósok körútja 2., 1117 Budapest, Hungary
e-mail: {zoltan.mann,szep.tamas.89}@gmail.com

Backtrack-style exhaustive search algorithms for NP-hard problems tend to have large variance in their runtime. This is because "fortunate" branching decisions can lead to finding a solution quickly, whereas "unfortunate" decisions in another run can lead the algorithm to a region of the search space with no solutions. In the literature, frequent restarting has been suggested as a means to overcome this problem. In this paper, we propose a more sophisticated approach: a best-first-search heuristic to quickly move between parts of the search space, always concentrating on the most promising region. We describe how this idea can be efficiently incorporated into a backtrack search algorithm, without sacrificing optimality. Moreover, we demonstrate empirically that, for hard solvable problem instances, the new approach provides significantly higher speed-up than frequent restarting.

Keywords: best-first search, backtrack, branch-and-bound, constraint satisfaction problem, frequent restarting.

1. Introduction

All known exact algorithms for NP-hard problems have super-polynomial (usually exponential) worst-case complexity. Luckily, smart algorithms are usually much faster on many practical problem instances than their worst-case complexity. However, this improved performance usually comes at a price: extremely high variability in running time. That is, the running time of an algorithm may vary dramatically (by multiple orders of magnitude) between runs on similar problem instances, or even between different runs on the same problem instance (Gomes *et al.*, 2000; Cheeseman *et al.*, 1991; Hogg and Williams, 1994; Jia and Moore, 2004). This high variability in algorithm runtime poses a significant challenge on the practical application of the algorithm, because it is hard to predict if the algorithm will solve a given problem instance within a couple of seconds or run for several days (or even longer).

Such smart exact algorithms for NP-hard problems are often variants of *backtrack search*, which operates on partial solutions, assigning values to variables one by one. When the algorithm can deduce that the current partial solution cannot be extended to a solution, then it backtracks, thus pruning a part of the search space. This pruning is very helpful in making the algorithm efficient enough even for many large problem instances.

Backtrack search algorithms also suffer from extreme variability in running time, especially on solvable instances. Intuitively, this is because "lucky" branching decisions can lead to finding a solution quickly, whereas "unlucky" decisions in another run can lead the algorithm to a region of the search space with no solutions.

A possible remedy for this issue that has been suggested in the literature is frequent restarting (Gomes *et al.*, 1998). If an algorithm involves random choices, it may make sense to run it several times on a given problem instance. For example, suppose that the median runtime of a randomized algorithm on problem instances of a given size is 1 minute. Assume that the algorithm has been running on a problem instance for 5 minutes without any results yet. Intuitively, one could think that the algorithm will most probably finish very soon, so we should keep waiting. However, empirical evidence shows that it is better to stop the current run of the algorithm and restart it. The rationale is that it might actually happen with surprisingly high probability that the current run of the algorithm will take several hours, days, or even longer. On the other hand, if we restart the algorithm, chances are high that the next run will be more fortunate and may finish in a minute or even less.

Thinking of a backtrack search, the reason why frequent restarting improves the performance of the algorithm is that this way long useless searches in areas of the search space with no solutions are stopped; the restarted search might be more lucky and find its way directly to a more promising part of the search space.

Although restarting works quite well in practice, it is a very simplistic approach to solve the inefficiencies of backtrack search. In a way, it is a brute-force approach: there is no guarantee whatsoever that the new run will be better. Instead, the rationale is that among several runs of the algorithm, there will be probably a lucky one.

In this paper, we propose a more sophisticated approach. We observed that the problem with backtrack search is rooted in its depth-first-search nature. This is why it cannot "give up" searching an unfruitful part of the search space and move on to other, more promising areas. Therefore, we propose to implement backtrack search with a best-first-search (BFS) heuristic that will guide it to different parts of the search space, always aiming for the most promising area. The modified algorithm is also an exact algorithm. If there is no solution, the modified algorithm will also perform a complete search to prove unsolvability. If the problem instance is solvable, then the modified algorithm is also guaranteed to find a solution, but in many cases it can find it much faster than a normal backtrack algorithm.

The rest of the paper is organized as follows. First, Section 2 reviews previous work on speeding up backtrack search algorithms. Next, we give a formal description of the problem domain that our algorithms attack: constraint satisfaction problems (Section 3), followed by a description of our reference backtrack algorithm (Section 4). This algorithm already contains several enhancements to make it smart and efficient. The main contribution of the paper, the extension of the backtrack algorithm with the BFS heuristic, is presented in Section 5. As it turns out, some of the improvement techniques contained in the reference backtrack algorithm to speed it up, most notably conflict-driven backjumping, make the extension with BFS quite tricky. We also prove the correctness of the algorithm. Next, Section 6 gives an example to facilitate the understanding of the presented algorithm. Section 7 presents the results of empirical measurements to compare the efficiency of the reference backtrack algorithm, backtracking accelerated with frequent restarting, and backtracking accelerated with BFS. Finally, Section 8 concludes the paper.

2. Previous work

Well-known early applications of the backtrack algorithm include the Davis–Putnam–Logemann–Loveland (DPLL) algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form (Davis and Putnam, 1960; Davis *et al.*, 1962), as well as Randall Brown's algorithm and its refinements for graph coloring (Brown, 1972; Brélaz, 1979).

With the wide-spread application of backtrack algorithms, researchers started to gain empirical experience with such algorithms in practice. The problem of the extremely high variability in algorithm runtime came up quickly, as documented, for example, by Knuth (1975): "Sometimes a backtrack program will run to completion in less than a second, while other applications of backtracking seem to go on forever. The author once waited all night for the output from such a program, only to discover that the answers would not be forthcoming for about 10^6 centuries." This motivated Knuth to devise methods to estimate the runtime of a backtracking algorithm using sampling strategies. Ever since, the prediction of algorithm runtime has remained an important and challenging research topic (see, e.g., the work of Hutter *et al.* (2006) for more recent results).

The fact that backtracking is much faster on many typical problem instances than its worst-case complexity spawned also interest in the rigorous mathematical analysis of the algorithm's complexity on random problem instances. For instance, Wilf (1984) showed that, for the graph coloring problem, the average-case complexity of backtrack search is only $O(1)$, in significant contrast to its exponential worst-case complexity. Through subsequent results, the behaviour of backtrack search on graph coloring is quite well understood (Bender and Wilf, 1985; Jia and Moore, 2004; Mann and Szajkó, 2010b; 2010a).

Of course, researchers also devised techniques to speed up backtrack search (Russell and Norvig, 2010). Some of the most important techniques include the following:

- The algorithm has some freedom in choosing the next variable to branch on as well as in determining the order in which the possible values will be assigned to that variable. By using appropriate heuristics for these choices, the algorithm can be made more efficient (Geelen, 1992).

- If the search problem exhibits symmetries, symmetry breaking techniques can be used to avoid searching equivalent portions of the search space, thus making the search more efficient without threatening optimality (Brown *et al.*, 1996).

- Backjumping aims to increase the size of the pruned part of the search space after a conflict by carefully analyzing the causes of the conflict and jumping back in the search tree potentially multiple levels (Dechter and Frost, 2002).

- Nogood learning can be used to record combinations of decisions that necessarily lead to a conflict, so that the same combination can be avoided in the future, preventing the exploration of certainly useless parts of the search space (Dechter, 1990).

- Consistency propagation techniques (arc consistency, i-consistency, etc.) make it possible to infer without branching that a variable cannot take one or more of its possible values, thus keeping the search tree relatively small (Dechter, 2003).

All these techniques make backtrack search smarter, so that it will be as efficient as possible on as many inputs as possible. However, even with these improvements, the algorithm's worst-case complexity remains exponential, so that the problem of high variability remains.

A different approach was used by Gomes *et al.* (2000), introducing the notion of "heavy-tailed distributions" to characterize the runtime distribution of typical backtrack algorithms for combinatorial problems. A heavy-tailed runtime distribution formalizes the experience that runs of an exact algorithm for an NP-hard problem often take much longer than the median runtime of the algorithm. Besides providing statistical description of such distributions, the authors suggested that the problem can be mitigated using frequent restarts of the algorithm. In fact, they suggest that deterministic algorithms should also be randomized in order to capitalize on the acceleration opportunity offered by frequent restarting (Gomes *et al.*, 1998).

Since then, frequent restarts have become an integral part of most successful solvers, e.g., Chaff (Moskewicz *et al.*, 2001) and MiniSAT (Eén and Sörensson, 2004). Also, several different schemes have been suggested concerning the frequency with which restarts should be carried out, respectively how the restart times should be increased during the course of the algorithm (Luby *et al.*, 1993; Biere, 2008; Kautz *et al.*, 2002). More recently, Haim and Heule (2010) pointed out that the optimal restart strategy is strongly dependent on the use of other improvement techniques (e.g., constraint learning), and, as a result of the trends in those other improvement techniques, optimal restart times decrease with newer solver generations.

The technique presented in this paper is comparable with frequent restarts in that it also addresses specifically the problem of heavy tails, i.e., the fact that unfortunate branching decisions early on in a backtrack search can easily lead to extremely long runs of the algorithm. However, our technique is more sophisticated and offers the possibility to resume a paused search instead of abandoning it completely. This way, we do not waste computational power as frequent restarts do.

Another related work is that of Schaefer *et al.* (2012) on hierarchical genetic search (HGS), which is—similarly to our approach—also a way of running multiple solver instances concurrently. However, the details of the two approaches are quite different, given that HGS operates with genetic algorithms whereas we use an exact algorithm for constraint satisfaction as the base solver. For example, HGS is characterized by a tree of populations with increasing accuracy; such a concept is not necessary in our method, making the interoperation between the solver instances simpler in our case.

3. Problem formulation

We consider the constraint satisfaction problem (CSP) as a general application domain of backtrack search. The volume of past research on applying backtrack search to a CSP and the fact that many combinatorial problems can be formulated in a natural way as a CSP make it an ideal testbed for our investigations. Also other popular problems, such as satisfiability or integer programming, can be easily turned into a CSP (Mann, 2011).

A CSP is defined on a set of variables $X = \{x_1, x_2, \ldots, x_n\}$. The domain of x_i is a finite, non-empty set denoted by $D(x_i)$ or D_i, consisting of the possible values for variable x_i.[1] We write $D := D_1 \times D_2 \times \ldots \times D_n$. A possible assignment of values to the variables is a vector $(a_1, a_2, \ldots, a_n)^T \in D$, assigning to each variable x_i a value $a_i \in D_i$.

We are also given a set of constraints $C = \{C_1, C_2, \ldots, C_m\}$. Each C_j is a pair (V_j, R_j), consisting of a subset of the variables $V_j \subseteq X$ and a relation R_j. If $V_j = \{x_{j_1}, x_{j_2}, \ldots, x_{j_k}\}$, then $R_j \subseteq D_{j_1} \times D_{j_2} \times \ldots \times D_{j_k}$. Here R_j defines which tuples of possible values of the involved variables satisfy the given constraint. Specifically, the assignment $(a_1, a_2, \ldots, a_n)^T \in D$ satisfies the constraint (V_j, R_j), where $V_j = \{x_{j_1}, x_{j_2}, \ldots, x_{j_k}\}$, iff $(a_{j_1}, a_{j_2}, \ldots, a_{j_k})^T \in R_j$. If the assignment does not satisfy the given constraint, then there is a *conflict* among the assignments $x_{j_1} = a_{j_1}, x_{j_2} = a_{j_2}, \ldots, x_{j_k} = a_{j_k}$.

The aim is to assign to each variable a value from its domain, such that all constraints are satisfied. That is, a *solution* is an assignment $(a_1, a_2, \ldots, a_n)^T \in D$, such that for all $(V_j, R_j) \in C$, if $V_j = \{x_{j_1}, x_{j_2}, \ldots, x_{j_k}\}$, then $(a_{j_1}, a_{j_2}, \ldots, a_{j_k})^T \in R_j$. The goal of the CSP is to decide whether a solution exists. If so, then the given CSP instance is *solvable*, otherwise it is *unsolvable*.

We will use the following simple CSP instance as a running example in the paper:

Variables: $X = \{x_1, x_2, x_3, x_4\}$
Domains: $D_1 = D_2 = D_3 = D_4 = \{0, 1\}$
Constraints:

$$C_1 : \quad x_1 < x_3 + x_4$$
$$C_2 : \quad x_2 + x_3 = x_1 \cdot x_3$$

[1] In the following examples, the D_i sets will consist of numbers, but this is not necessary.

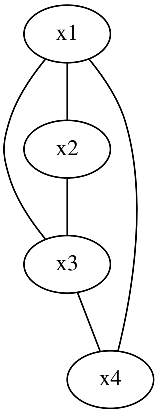


Fig. 1. Constraint graph of the example CSP.

This CSP instance is solvable, and a solution is, for example, $x_1 = x_2 = x_3 = 0, x_4 = 1$.

The *constraint graph* of a CSP is an undirected graph $G = (X, E)$, in which the vertices are the variables. Two variables x_i and x_j are adjacent if there is a constraint containing both of them. The constraint graph of the above example is shown in Fig. 1. The constraint graph allows us to speak about, e.g., the "neighbours of a variable" or the "constraints incident to a variable".

It will also be useful to consider partial assignments, in which some variables are assigned a value whereas others are unassigned. The notation $x_i = \varepsilon$ means that x_i is unassigned. We assume that ε is a new symbol, not contained in any D_i. We use $D'_i := D_i \cup \{\varepsilon\}$ and $D' := D'_1 \times D'_2 \times \ldots \times D'_n$. Then, a *partial assignment* is simply an element of D'. If it is also an element of D, i.e., the value ε is not used, then it is a *full assignment*.

Given a partial assignment $v = (a_1, a_2, \ldots, a_n)^T \in D'$, the set of assigned variables will be denoted by $A(v) = \{x_i \in X : a_i \neq \varepsilon\}$, the set of unassigned variables by $U(v) = \{x_i \in X : a_i = \varepsilon\}$. A partial assignment also allows a subset of the constraints to be evaluated, namely, those in which only the assigned variables participate. That is, the set of *evaluable constraints* is $EC(v) = \{(V_j, R_j) \in C : V_j \subseteq A(v)\}$. The partial assignment v is *0-consistent* if it satisfies all evaluable constraints.

In the running example, consider the partial assignment $v = (0, 0, 0, \varepsilon)$. Then, $A(v) = \{x_1, x_2, x_3\}$, $U(v) = \{x_4\}$, and $EC(v) = \{C_2\}$. Since $(x_1, x_2, x_3) = (0, 0, 0)$ satisfies constraint C_2, v is 0-consistent.

4. Backtrack algorithm

4.1. Basic version of the algorithm. The algorithm assigns values to the variables, one at a time, as long as no conflict occurs. If all variables can be assigned a value, a solution is found and the algorithm terminates. If there is a conflict, the algorithm backtracks, i.e., it goes back to the last consistent state by undoing the last decision. Then it proceeds to an unexplored branch by trying a new value for the currently selected variable. When all possible branches from a given state have been tried without success, the algorithm backtracks further.

The algorithm traverses the partial assignments in a tree structure. There are two possible termination situations: either a solution is found, or the algorithm checks all branches from the root of the tree without success, and tries to backtrack from the root. In this case, we can be sure that the input problem instance is unsolvable. Clearly, the algorithm terminates in finite time, since the size of the complete search tree is an upper bound on the number of steps of the algorithm. This number is exponentially high, but in many cases the algorithm can prune large subtrees of the search tree, which can considerably decrease its runtime.

More formally, a *complete search tree* is a directed tree T with the following characteristics:

- Each node of T is a partial assignment.
- The root of T is the partial assignment $(\varepsilon, \varepsilon, \ldots, \varepsilon)$.
- Each leaf of T is a full assignment. Moreover, each full assignment is one of the leaves, i.e., the number of leaves is $|D_1| \cdot |D_2| \cdot \ldots \cdot |D_n|$.
- Let $v = (a_1, a_2, \ldots, a_n)$ be an inner node of T. Then, there is a variable $x_i \in U(v)$ such that the children of the node v in T are $\{v' = (a'_1, a'_2, \ldots, a'_n) : a'_i \in D_i$ and $a'_j = a_j$ for all $j \in \{1, 2, \ldots, n\} \setminus \{i\}\}$.

The last point means that, at a given partial assignment v, the algorithm chooses an unassigned variable x_i and tries to assign all possible values to it; the resulting partial assignments will be the children of v. The unassigned variable chosen in node v is denoted by $x(v)$. The child of v that is obtained from v by assigning a to x_i (where $x_i = x(v)$ and $a \in D_i$) is denoted by $v[x_i \leftarrow a]$.

The complete search tree is not unique, as different choices of the unassigned variable to branch on will result in different complete search trees. However, all possible complete search trees have the same set of leaves, namely, all full assignments. A possible complete search tree of the above example is given in Fig. 2.

The algorithm does not have to visit all nodes of the complete search tree. There are two reasons for this:

- If the algorithm reaches a non-0-consistent partial assignment, then it can backtrack, because this partial assignment surely cannot be extended to a solution. For instance, the partial assignment

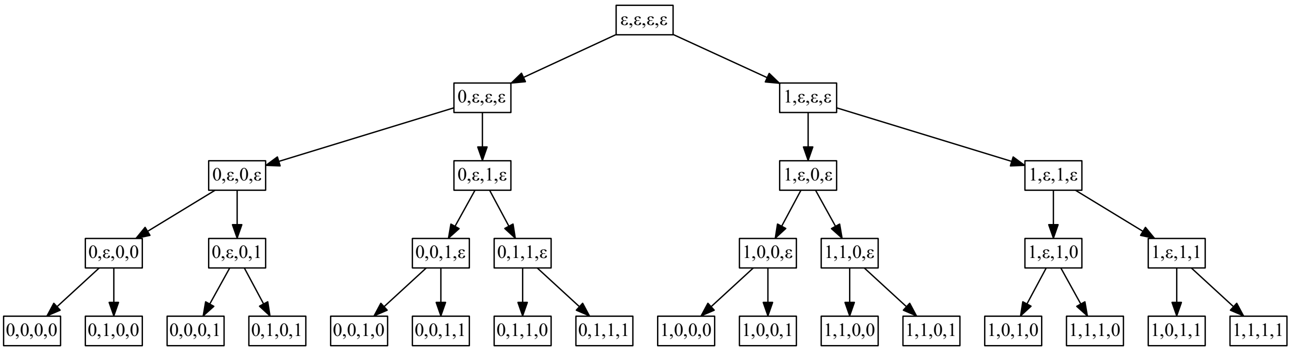


Fig. 2. Complete search tree of the example CSP.

$(0, \varepsilon, 0, 0)$ in the above example is not 0-consistent because the constraint C_1 is not fulfilled. Therefore, the subtree under this partial assignment can be pruned.

- If the algorithm reaches a leaf that is a solution, then it can stop and return the solution found.

For these reasons, the *actual search tree* will be a subtree of T.

Algorithm 1. Basic backtrack algorithm.

```
$v := (\varepsilon, \varepsilon, \ldots, \varepsilon)$
**while** true **do**
    **if** $v$ is not 0-consistent **then**
        BACKTRACK
    **else if** $v$ is a full assignment **then**        //v is a solution
        return $v$
    **else**        //v is 0-consistent, but not a full assignment
        choose an unassigned variable $x_i$ and let $x(v) := x_i$
        choose a value $a$ from $D_i$
        change $v$ by assigning $a$ to $x_i$        //move to a child node
    **end if**
**end while**

**procedure** BACKTRACK
**repeat**
    **if** $v$ is the root of $T$ **then**
        return UNSOLVABLE
    **else**
        change $v$ by letting $x(v) = \varepsilon$        //move to the parent node
    **end if**
**until** $v$ has children that are not visited yet
change $v$ by assigning a new value to $x(v)$   //move to next child node
**end procedure**
```

The basic version of the backtrack search algorithm is described in pseudo-code in Algorithm 1.

4.2. Improvement techniques used.

Beyond the basic backtrack algorithm described above, we used a number of techniques to make it competitive, as shown next.

4.2.1. Consistency propagation.

In an interim state of the algorithm, we are given a 0-consistent partial assignment v, meaning that values of the assigned variables satisfy all evaluable constraints. However, the values of the assigned variables can also impact the possible values of the unassigned variables. Through intelligent use of this information, we can detect earlier that the partial assignment is bound to lead to a conflict.

Consistency propagation in our algorithm combines two techniques: 1-consistency and arc-consistency. In order to define 1-consistency, let x_i be an unassigned variable, and let $b \in D_i$ be a possible value of x_i. Then, b is a *1-consistent value* for x_i, if $v[x_i \leftarrow b]$ is still 0-consistent. Moreover, v is *1-consistent*, if there is a 1-consistent value for all unassigned variables $x_i \in U(v)$. Obviously, non-1-consistent values of an unassigned variable will lead to a conflict; hence, if v is not 1-consistent, then it cannot be extended to a solution.

In order to maintain 1-consistency, we store for each unassigned variable $x_i \in U(v)$ its set of 1-consistent values, denoted by $\Delta_v(x_i)$, or simply $\Delta(x_i)$ if there is no ambiguity about v. At the beginning, $\Delta(x_i)$ is initialized to D_i. Later on, every time a value is assigned to a variable x_j, all constraints involving x_j are examined to check if it can be inferred that a value of another, unassigned variable x_i has become non-1-consistent, and if so, it is removed from $\Delta(x_i)$. If $\Delta(x_i)$ becomes empty for an unassigned variable x_i, then we backtrack.

If $\Delta_v(x_i) = \{a\}$, then obviously, in any extension of v to a full assignment, $x_i = a$, so we can perform the same steps as above to maintain 1-consistency, as if x_i were already assigned the value a, thus possibly removing further non-1-consistent values of other unassigned variables. This is *arc-consistency* propagation.

To see the power of consistency propagation, take the partial assignment $v = (0, \varepsilon, \varepsilon, 0)$ in our running example. Since x_1 and x_4 already have fixed values, constraint C_1 can be examined to check which values of x_3 are 1-consistent. The value 0 is not 1-consistent for x_3, yielding $\Delta_v(x_3) = \{1\}$. As $\Delta_v(x_3)$ contains only one element, we can use arc-consistency propagation and analyze constraint C_2, in which now x_1 and x_3 already have fixed values. None of the possible values of x_2 are 1-consistent, and hence v cannot be extended to a solution: we can backtrack and prune the whole subtree under v.

4.2.2. Variable selection. For choosing the next variable to branch on, we use the MRV (minimum remaining values) heuristic, selecting the variable with the least remaining values in its domain, i.e., we choose the unassigned variable x_i for which $|\Delta(x_i)|$ is minimal. This helps in keeping the size of the tree as small as possible. To break ties, we use the degree heuristic, choosing the variable with the highest number of unassigned neighbours. This helps the consistency propagation mechanism to infer as much information as possible concerning the neighbouring variables. If there is still a tie, we choose the variable with the lowest index.

4.2.3. Value selection. After choosing a variable to branch on, the value selection heuristic defines in which order the child nodes are visited. If the problem instance is not solvable, then all children of the current node must be visited anyway, until the algorithm finally backtracks from this partial assignment. Hence, in this case, the value selection heuristic plays no important role. But if the problem instance is solvable, then it may make the algorithm significantly faster if it chooses the right value first. Therefore, the aim of the value selection is to propose the most promising values first. For this reason, we start with the value that constrains the neighbouring unassigned variables the least. More formally, if x is the chosen variable to branch on, a is a possible value for x, and y is an unassigned neighbour of x, then let $\lambda(a, y)$ denote the number of values in $\Delta(y)$ that would have to be removed from $\Delta(y)$ if a were assigned to x. Furthermore, let $\lambda(a) := \sum_{y \in N(x) \cap U(v)} \lambda(a, y)$, where $N(x)$ is the set of neighbours of x. In other words, $\lambda(a)$ is the total number of values that will be removed—as a result of the constraints—from the neighbouring unassigned variables. We assign the values in increasing order of λ.

4.2.4. Unimportant variables. Let x be an unassigned variable. If we can be sure that $\Delta(x)$ will be non-empty after any consistent assignment of the remaining variables then x is an *unimportant variable*, and can be removed without affecting solvability.

Two simple examples are the following. If a variable has more possible values in its Δ set than the number of values that the remaining (not yet satisfied) constraints can possibly remove, then it is an unimportant variable, because we can surely satisfy all of its constraints.

In the second case, the variable x is unimportant if there is another variable y, such that $\Delta(y) \subseteq \Delta(x)$, and in each constraint involving x, if x is replaced by y, an existing constraint of y is obtained. In this case, x is indeed unimportant, because assigning the same value to x as to y will satisfy all constraints involving x, if all constraints involving y can be satisfied.

4.2.5. Conflict-driven backjumping. Suppose that, during the course of the algorithm, the partial assignment $v[x \leftarrow a]$ has been visited and the algorithm found out that this partial assignment cannot be extended to a solution; hence, the algorithm backtracks to the partial assignment v, and will try another child $v[x \leftarrow a']$ next. However, sometimes it can be established that failure of the partial assignment $v[x \leftarrow a]$ was *not* due to x having the value a, but rather to other decisions encoded within v. In this case, other children of v will also definitely fail, so there is no point in visiting them. Rather, we should backjump directly to the last node of the search tree that is not guaranteed to fail, by undoing one of the decisions responsible for the failure of $v[x \leftarrow a]$.

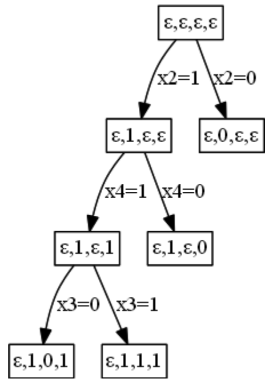


Fig. 3. Case for conflict-driven backjumping.

To demonstrate this phenomenon, we revisit our running example. Consider the situation depicted in Fig. 3, in which the algorithm made the following assignments (in this order): $x_2 \leftarrow 1, x_4 \leftarrow 1, x_3 \leftarrow 0$. At this point, the consistency propagation mechanism will remove both values from $\Delta(x_1)$, because none of them fulfils constraint C_2. Therefore, the algorithm backtracks and tries the other possible value for x_3, thereby reaching the partial assignment $(\varepsilon, 1, 1, 1)$. Again, the consistency propagation mechanism will remove both the values from $\Delta(x_1)$, because none of them fulfils the constraint C_2. Therefore, the algorithm will backtrack again: it will undo the $x_3 \leftarrow 1$ decision. Then it recognizes that the partial assignment $(\varepsilon, 1, \varepsilon, 1)$ has no unexplored children, hence it goes back to the partial assignment $(\varepsilon, 1, \varepsilon, \varepsilon)$ and then to its unexplored child $(\varepsilon, 1, \varepsilon, 0)$.

However, if we look more carefully at the *reasons* for the failure of the abandoned branch of the tree, we can conclude that the choice $x_4 \leftarrow 1$ did not contribute to the failure. Rather, the former choice $x_2 \leftarrow 1$ led to a situation where both possible values of x_3 became infeasible. This is easy to see, because only constraint C_2 played a role in establishing the conflict, in which x_4 does not appear. Therefore, the newly visited branch, which differs from the old one only in the value of x_4, is also doomed for failure. In other words, we can backjump directly to the partial assignment $(\varepsilon, \varepsilon, \varepsilon, \varepsilon)$ by undoing the unlucky choice of x_2 and trying a new value for it.

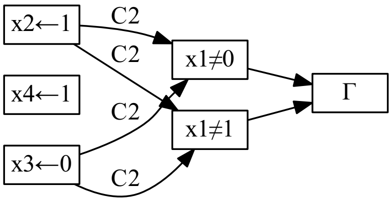


Fig. 4. Example constraint effect graph.

In general, to know the reasons for failure, we use the *constraint effect graph*. Each node in this graph represents an action that the algorithm has carried out: either assigning a value to a variable or removing a value from a variable's Δ set. In addition, there is a special node, denoted by Γ, representing failure. Every time the algorithm assigns a value to a variable, it creates an appropriate new node in the constraint effect graph. Every time the algorithm removes a value from $\Delta(x)$ for a variable x, an appropriate node is created and connected by a directed edge with all other nodes that are reasons for this action; the edges are directed towards the new node. If $\Delta(x)$ becomes empty, the corresponding nodes are connected with Γ by a directed edge, pointed towards Γ. An example can be seen in Fig. 4, showing the conflict situation after the assignments $x_2 \leftarrow 1, x_4 \leftarrow 1, x_3 \leftarrow 0$.

With this technique in place, the reason for failure can be found: it is the set of assignments of values to variables from which there is a directed path to Γ in the constraint effect graph. We must differentiate between two backtrack situations:

- We backtrack from v because consistency propagation emptied $\Delta(x)$ for some variable x, meaning that there is no value for x that would be consistent with the values chosen for its neighbours. We will call such a backtrack a *direct backtrack*.
- We backtrack from v because all children of v have been visited without finding a solution, i.e., we have backtracked already from the last child of v. We will call such a backtrack an *indirect backtrack*.

Backjumping is only possible in the case of an indirect backtrack. To see this, assume that we make a direct backtrack from $v[x \leftarrow a]$ to v. This means that v was a consistent state, but setting $x \leftarrow a$ made it inconsistent. Then, the reasons for failure in state $v[x \leftarrow a]$ include the setting made for variable x; therefore, changing x to another value might resolve the conflict.

Now assume that the algorithm is about to make an indirect backtrack from state v, because all children of v have been visited in vain. Then it is important to know the set of decisions (assignments of values to variables) that contributed to the failure of the children of v. Let v' be a child of v. At the moment when the algorithm was about to backtrack from v', the algorithm was able to determine, using the constraint effect graph, the set of variables whose assignment contributed to the failure of v'. We call this the *conflict set* of v' and denote it by $L(v')$. Then, the reason for failure of v is $\bigcup\{L(v') : v'$ is a child of $v\}$. It can be seen easily that, as long as all variables in this set retain their values, failure is guaranteed; however, if at least one of them changes its value, then there is a chance for success. This means that the algorithm can backjump as much as necessary to undo one of the decisions concerning these variables.

4.3. Determinism.

It is important to note that the algorithm that we presented so far—which we will call the *reference algorithm*— is fully deterministic. That is, the input problem instance determines what the complete search tree will look like, which of its nodes will be visited and in what order. To formalize this, let the *state* of the algorithm at any point during the run of the algorithm comprise all information that the algorithm has gathered and stores for future use. Specifically, the state, denoted by σ, consists of the following pieces of information:

- The current partial assignment v.
- The order in which the variables in $A(v)$ were assigned their values.
- For all $x \in A(v)$, the set of values that have already been tried for x.
- For all $x \in U(v)$, the set $\Delta_v(x)$.
- The set of unimportant variables.
- The conflict set of v and its ancestors in the tree.

The algorithm starts from an initial state σ_0, and in each step it moves to a new state. Since the state contains all information necessary to determine the next state, the state after step $i + 1$ is a function of the state after step i. That is, $\sigma_{i+1} = next(\sigma_i)$, where the function $next$ defines how the algorithm steps to the next state. It is important to note that the $next$ function induces a linear order among the visited nodes of the search tree.

5. Best-first-search

We now describe our proposed extension of the backtrack algorithm to overcome the limitations stemming from its depth-first-search (DFS) strategy. We extend it with a best-first-search (BFS) strategy, so that it can jump quickly between different parts of the search tree, always focusing on the most promising part of the tree.

In doing so, the biggest challenge is to make sure that, on the one hand, the algorithm remains complete and, on the other we do not waste time on visiting the same node of the search tree multiple times. Therefore, we must maintain in a memory-efficient fashion what nodes have already been visited. For this reason, we do not use a full-fledged BFS, which would allow full freedom in moving in the tree, but rather a combination of a controlled BFS and the more "disciplined" DFS inherent in the underlying backtrack algorithm.

Informally, the idea is that the algorithm should run as the reference algorithm would, but sometimes it can jump forward or backward in the search, so that it may find a solution faster. As an analogy, when looking for something in a book, we read a couple of sentences, then jump to another part, read there again some sentences, etc.

We implement this scheme by launching several copies of the reference algorithm at different points in the search tree. We start one of these search instances and let it run for a while. Afterwards, we pause this run and transfer control to another search instance, etc. The currently visited node of search instance S, denoted by $cn(S)$, is stored when it is paused, so that it can later be resumed from this node of the search tree.

In order to have full control over what part of the search tree each search instance is visiting, we confine all but one search instances to the subtree rooted in the node in which they were started. Specifically, we create a *main search instance* S^* and some *normal search instances* $S_1, S_2, \ldots, S_k$. For each search instance S, let $sn(S)$ denote the start node of S, and, let $T(S)$ denote the subtree of S, that is, the subtree of the search tree with root $sn(S)$. A normal search instance S_i is only allowed to search within $T(S_i)$, i.e., when it tries to backtrack from $sn(S_i)$, it is stopped. Moreover, for any two search instances S and S', $sn(S)$ must not be within $T(S')$ and vice versa, $sn(S')$ must not be within $T(S)$. We will call these rules the *search instance consistency rules*. Because of these restrictions, all normal search instances will scan disjoint parts of the search tree.

The main search instance S^* is not confined to $T(S^*)$. When it finishes scanning $T(S^*)$, it will also scan all parts of the search tree that are not covered by any other search instances, thereby guaranteeing the completeness of the algorithm. However, every time the main search instance moves downwards in the search tree, we must check if it reached the start node of a normal search instance. If it reaches $sn(S_i)$, then we know that the next steps of the search until the current position of S_i have already been visited by S_i, and thus S^* can move on from the current position of S_i. In this case, S_i and S^* must be merged. This step will be explained later on in detail.

Algorithm 2. Backtrack algorithm with BFS logic.

```
Create search instances $S^*$ and $S_1, S_2, \ldots, S_k$
Set the status of all search instances to active
while true do
    Pick an active search instance $S$
    Run $S$ for at most $N$ steps
    if $S$ found a solution then
        Return with the found solution
    else if $S \neq S^*$ and $S$ tried to backtrack from $sn(S)$ then
        Set $S$ to passive
    else if $S = S^*$ and it tried to backtrack from the root node then
        Return with UNSOLVABLE
    else if $S = S^*$ and it reached the start node of some $S_i$ then
        Merge $S$ and $S_i$
    end if
end while
```

Algorithm 2 shows the overall flow of the algorithm. The non-trivial details of the algorithm are described in the following subsections.

5.1. Creating search instances.

First, we define the starting partial assignments for the search instances. These are selected randomly from the nodes on the given level(s) of the search tree, except that we pay attention to the search instance consistency rules. Next, the creation of a search instance S is carried out by emulating the behaviour of the backtrack algorithm and steering it directly to the desired start node $sn(S)$ by assigning the chosen values to the variables in $A(sn(S))$. We repeat this for all search instances. Specifically, for creating the main search instance, we pick the first possible value for each variable, so that the start node of the main search will be in the left-most branch of the search tree. When creating the normal search instances, if we decide to emulate a forward step by assigning the i-th value to the next variable, we have to remove all of the preceding $i-1$ possible choices.

5.2. Status of a search instance.

The status of a search instance can be either *active* or *passive*. Each search instance is initialized as active and remains so until it finishes searching the subtree that it is confined to. That is, when search instance S tries to backtrack from $sn(S)$, it becomes passive. From this moment, S will not be run anymore. However, it cannot simply be removed, because it manifests the important information that $T(S)$ has already been scanned. In practical terms, the difference between an active and a passive search instance is that an active search instance can be picked for further running, while a passive one cannot.

The main search instance remains active during the whole algorithm.

5.3. Picking the search instance to run next. The search instance to run next is selected from the active search instances. The idea of best-first-search is to use a *valuation function* Q that lets us estimate the value of each active search instance (higher Q values are better). We choose with probability p the search instance with highest value, and with probability $1-p$ a search instance uniformly at random. The latter should help avoid the degeneration where for some reason always the same search instance is selected.

Our valuation function is defined in such a way that it favours search instances that are more likely to produce a solution. It is computed for search instance S as follows:

$$Q(S) = Q_1(sn(S)) + Q_2(cn(S)) + Q_3(S) + Q_4(S).$$

Q_1 and Q_2 judge how promising the start node, respectively the current node, of S is. They do this in the same way, but with potentially different coefficients: $Q_1(v) = c_1 q(v)$, $Q_2(v) = c_2 q(v)$, and $q(v) = |A(v)| \cdot \sum_{x_i \in U(v)} |\Delta_v(x_i)|$. That is, Q_1 and Q_2 favour nodes where the number of assigned variables is high and the unassigned variables have many possible values. In such cases, it is likely that we can choose suitable values for the unassigned variables.

Q_3 accounts for the number of steps that S has already made. If this number is high, it means that S did not really prove to be as good as it seemed because it failed to lead to a solution quickly. Thus, Q_3 should favour search instances with a low number of steps made, and hence we chose $Q_3(S) = c_3 \frac{1}{steps(S)}$.

Q_4 is based on the set of nodes that search instance S has already visited (denoted by $visited\_nodes(S)$). If S has already led to a node that is almost a solution, i.e., where almost all variables could be assigned a value without a conflict, then it seems more likely that it will indeed lead to a solution. Therefore, we chose $Q_4(S) = c_4 \max\{|A(v)| : v \in visited\_nodes(S)\}$.

5.4. Running a search instance. The selected search instance is executed and it emulates the steps of the underlying backtrack algorithm. The search instance is allowed to run for at most a given number of steps, where one step corresponds to one execution of the $next$ function described in Section 4.3. When the search instance S is first run, it starts from $sn(S)$. When it is stopped, its current node $cn(S)$ is stored so that the next time it can continue running from this node. S runs until one of the following happens:

- The given maximum number of steps has been made.
- A solution is found.
- $S \neq S^*$ and it tries to backtrack from $sn(S)$, meaning that $T(S)$ has been scanned without finding a solution.
- $S = S^*$ and it tries to backtrack from the root node, meaning that the whole search tree has been scanned without finding a solution.
- $S = S^*$ and it reached $sn(S_i)$ for some normal search instance S_i.

5.5. Merging two search instances. First, let us assume that we do not use conflict-driven backjumping. The more complex case when conflict-driven backjumping is also applied will be discussed in detail in Section 5.6.

If S^* reaches $sn(S_i)$, this means that all nodes from the initial node (the root of the search tree) until $sn(S_i)$ have been checked by S^* and all nodes between $sn(S_i)$ and $cn(S_i)$ have been checked by S_i. (Intervals of nodes are understood with respect to the linear order induced by the $next$ function.) That is, all nodes between the initial node and $cn(S_i)$ have been checked. In this case, S^* and S_i are merged into a single search instance, which will be in all aspects identical to S_i, but it will be the new main search instance. In practical terms, this means that we remove the old S^* and promote S_i to be the new S^*.

The new S^* continues the search from $cn(S_i)$. This is correct, as all nodes before $cn(S_i)$ have already been checked. The merged search will continue to scan the remainder of $T(S_i)$. Afterwards it goes on with the rest of the search tree, as this is now the main search instance. Hence, we do not lose completeness through the merge.

It might seem that removing the old S^* incurs a loss of information. However, the state of the old S^* when reaching $sn(S_i)$ was exactly the same as that of S_i when it had started in $sn(S_i)$. This is because—under the assumption that there is no conflict-driven backjumping—the state of a search instance is uniquely determined by its current location within the search tree. Therefore, we do not lose information, we just avoid redundancy. All information necessary for the merged search instance to continue its work is contained in the state of S_i.

5.6. Handling conflict-driven backjumping. In order to perform conflict-driven backjumping, the conflict sets must be maintained during the search. For this reason, the state of the search instance will not only depend on its current node, but also on previously visited nodes. This phenomenon is illustrated schematically in Fig. 5.

In such a case, when the reference algorithm backtracks from node Y, it establishes the conflict set $L(Y)$ containing the variable assignments that contributed to the failure at Y. Later, when backtracking from

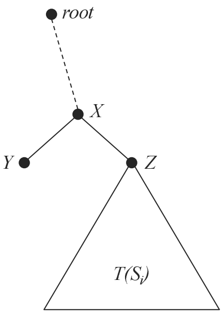


Fig. 5. Example for the effect of conflict-driven backjumping.

Z, $L(Z)$ is determined analogously, and then $L(X)$ is computed as $L(X) = L(Y) \cup L(Z)$, and this information is used to decide where to backjump from X.

Using multiple search instances makes this more complex. If the subtree rooted in Z is searched by search instance S_i, this has two important consequences:

- S_i can correctly determine the conflict sets for all nodes within $T(S_i)$ because the conflict sets are computed bottom-up, i.e., starting from the leaves and going upwards in the search tree.
- S_i does not know $L(Y)$ and so would not be able to correctly determine the conflict set of X or other nodes outside of $T(S_i)$.

That is, as long as S_i is confined to $T(S_i)$, it will behave exactly as the reference algorithm does. With S^*, there is also no problem because it starts in the very first branch and works its way exactly as the reference algorithm. The only problem occurs when S_i is merged with S^* and should continue working outside of $T(S_i)$. Merging takes place when S^* goes from X to Z. At this point, S^* has already determined the conflict set for all nodes until this node, including $L(Y)$. $L(Z)$ has been determined or will be determined (depending on whether S_i is still active or not) by S_i. The two search instances together have all information necessary to compute $L(X)$ and later also the conflict sets of the other nodes. To conclude: when merging S^* and S_i, the conflict information maintained by the two search instances must be united in the new S^*, and this way we get a search instance that has the same information as the reference algorithm would at the same point.

Conflict-driven backjumping may also lead to another interesting phenomenon. Looking again at Fig. 5, it can happen that S^*, when backtracking from Y (or one of its descendants), infers that it can actually backjump directly to an ancestor of X in the search tree. As a consequence, S^* will not visit node Z, and thus it will never be merged with S_i. This actually means that the subtree rooted at Z is guaranteed to not contain any solutions, and S^* managed to infer this without the need for S_i to scan the subtree. In this case, running S_i is a waste of time, but otherwise it does no harm. Since this phenomenon happens rarely, we decided to simply accept it. However, it should be noted that, if the given problem instance is unsolvable, then the steps taken by such useless search instances are the *only* overhead (in terms of the number of backtracks) of the best-first-search algorithm compared to the underlying backtrack algorithm. Our empirical results suggest that this overhead is minimal.

5.7. Correctness. Since we presented our algorithm in a semi-formal way, this allows a semi-formal proof of its correctness. We assume that the reference algorithm, which we will denote with A_{ref}, is correct. Based on this, we prove that the algorithm extended with the best-first-search logic, denoted by A_{BFS}, is also correct.

First, we introduce some notation and conventions. Let v_1 and v_2 be two nodes of the complete search tree. We write $v_1 \prec v_2$ if v_1 precedes v_2 in the depth-first-search order inherent in A_{ref}. This is almost the same as the linear order induced by the $next$ function, with the only difference that $next$ is only defined for the nodes that are actually visited by the algorithm, whereas $\prec$ is defined for all node pairs of the complete search tree, i.e., also for nodes that are skipped by the algorithm. We will assume that the search instances of A_{BFS} are indexed such that $sn(S_1) \prec sn(S_2) \prec \ldots \prec sn(S_k)$.

Proposition 1. *Let v be a node of the search tree that is visited by A_{ref}. Then, A_{BFS} will either also visit v or it will find a solution earlier.*

Proof. We differentiate between two cases.

Case 1: There is an i such that $v \in T(S_i)$. In this case, as long as v is not yet visited, S_i remains in an active status, and will be selected to be run again and again. Since within $T(S_i)$, S_i does the same as A_{ref}, it will visit v after a finite number of steps.

Case 2: v is in no $T(S_i)$. Let j denote the number of search instances whose start node lies before v according to $\prec$, i.e. $sn(S_j) \prec v \prec sn(S_{j+1})$. We prove the claim using induction according to j. If $j = 0$, then similarly as in Case 1, S^* will visit v after a finite number of steps.

Now, assume that the claim is already proven for all $j' < j$. We must again differentiate between two sub-cases.

Case 2.1. $sn(S_j)$ is visited by A_{ref}. In this case, the parent node of $sn(S_j)$, denoted by v', is also visited by A_{ref}. Moreover, $sn(S_{j-1}) \prec v' \prec sn(S_j)$ and thus, according to induction, v' is also visited by A_{BFS}, specifically by S^*. When S^* goes from v' to $sn(S_j)$, it is merged with S_j, and continues the search from $cn(S_j)$

with exactly the same state as A_{ref}. Since $sn(S_j) \prec v$, $v \notin T(S_j)$ and $cn(S_j) \in T(S_j)$, it follows that $cn(S_j) \prec v$. From this point, S^* will visit v after a finite number of steps, similarly to Case 1.

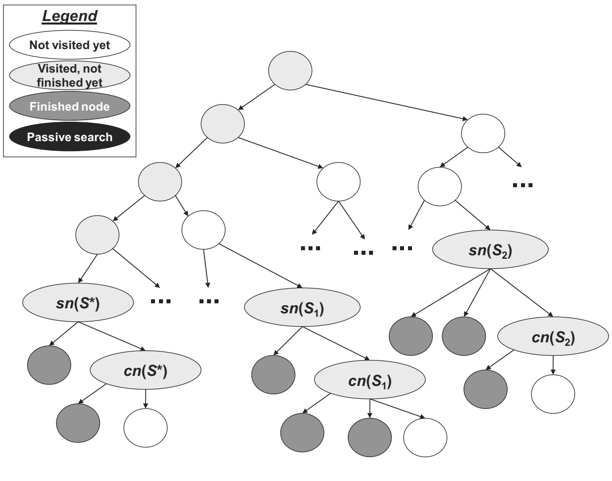
Case 2.2. $sn(S_j)$ is not visited by A_{ref}. In this case, let v' denote the last node before $sn(S_j)$ that was visited by A_{ref}. Since $v' \prec sn(S_j)$, according to induction, v' is also visited by A_{BFS}. Moreover, when S^* visits v', it has the same state as A_{ref}. Therefore, the next node visited by A_{ref}, $v'' = next(v')$, is also the next node visited by S^*. Since v is visited by A_{ref}, $v'' \prec v$ must hold. From this point, S^* will visit v after a finite number of steps, similarly to Case 1. ■

Corollary 1. *If A_{ref} always returns a correct answer after a finite number of steps, then so does A_{BFS}.*

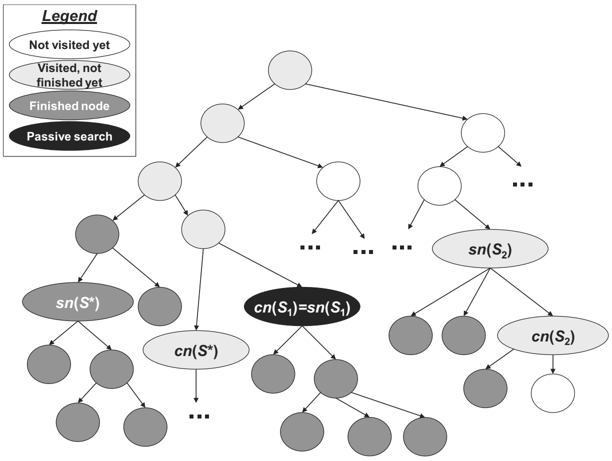
Proof. Since the complete search tree is finite and A_{BFS} visits each node of the search tree at most once, it will clearly return a result after a finite number of steps. If the given problem instance is not solvable, then of course A_{BFS} will not be able to find a solution and hence it will correctly return UNSOLVABLE. If the problem instance is solvable, then, according to our assumption, A_{ref} will find a solution. According to the Proposition, A_{BFS} will also correctly find this, or another, solution. ■

6. Example

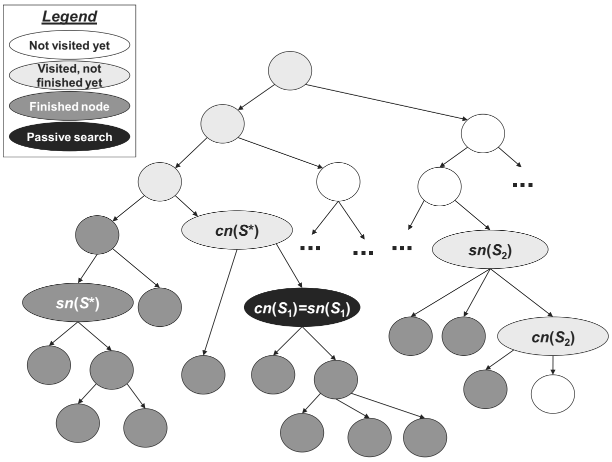
In order to make the operation of the algorithm clearer, we demonstrate it on a simple schematic example. We assume an unsolvable problem instance, and use 3 search instances: a main search instance S^* and 2 normal search instances S_1, S_2. First, we let them run for some time:



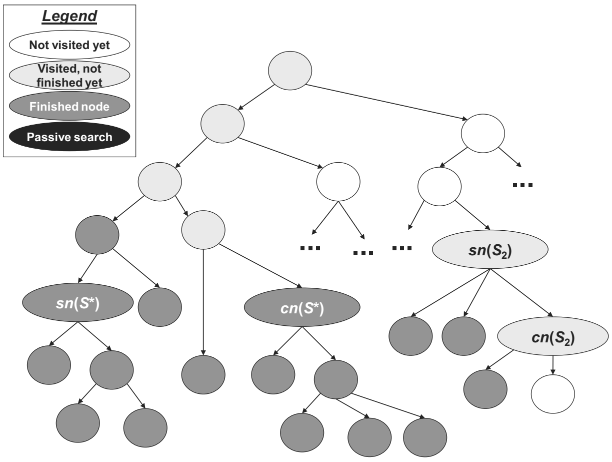
The main search instance finishes its original branch and searches further in the search tree. The search instance 1 finishes its own subtree, so it is not allowed to search further:



Then, the main search instance arrives to the start node of the search instance 1:



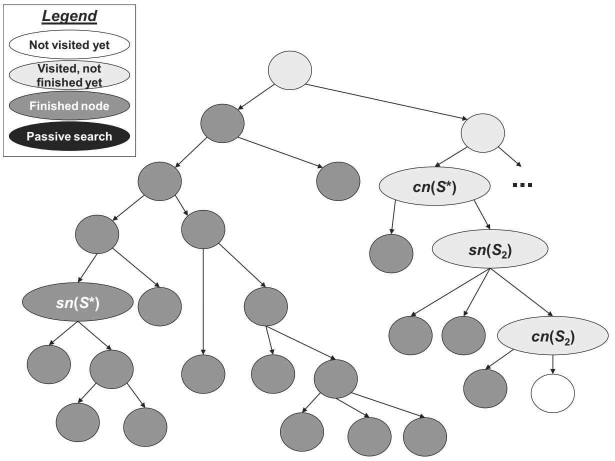
Now, the main search instance is merged with the search instance 1:



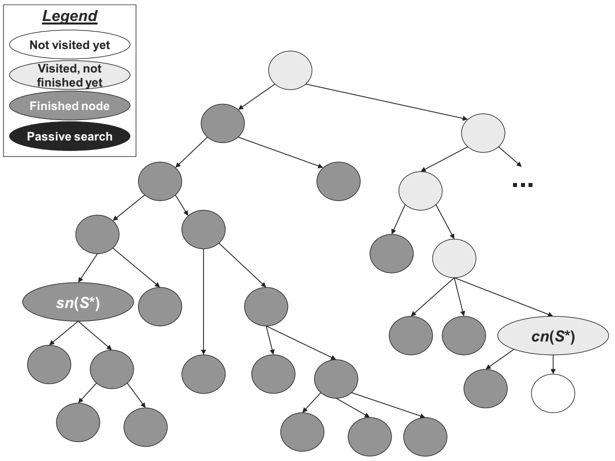
The main search instance continues to work its way until it arrives to the start node of the search instance 2:

Table 1. Properties of the structured instances used. The number of variables is denoted by n, the number of constraints by m. RAP means the register allocation problem, FAP means the frequency assignment problem.

Name	n	m	Source
queen7x7	49	476	DIMACS benchmark (Trick, 2003)
queen8x8	64	728	DIMACS benchmark (Trick, 2003)
queen9x9	81	2112	DIMACS benchmark (Trick, 2003)
queen10x10	100	2940	DIMACS benchmark (Trick, 2003)
ash331GPIA	662	4185	DIMACS benchmark (Trick, 2003)
ash608GPIA	1216	7844	DIMACS benchmark (Trick, 2003)
ash958GPIA	1916	12506	DIMACS benchmark (Trick, 2003)
school1	385	19095	DIMACS benchmark (Trick, 2003)
school1_nsh	352	14612	DIMACS benchmark (Trick, 2003)
RAP1792	334	14157	RAP (Appel and George, 1996)
RAP3678	380	11621	RAP (Appel and George, 1996)
RAP0040	580	12082	RAP (Appel and George, 1996)
RAP24050	337	10612	RAP (Appel and George, 1996)
RAP24467	420	13804	RAP (Appel and George, 1996)
RAP16616	845	14189	RAP (Appel and George, 1996)
RAP22950	621	9200	RAP (Appel and George, 1996)
FAP01	703	16438	FAP (Mann and Szajkó, 2012)
FAP02	481	11211	FAP (Mann and Szajkó, 2012)
FAP03	360	7756	FAP (Mann and Szajkó, 2012)
FAP04	434	10056	FAP (Mann and Szajkó, 2012)
FAP05	409	9401	FAP (Mann and Szajkó, 2012)
FAP06	385	8422	FAP (Mann and Szajkó, 2012)
FAP07	385	10278	FAP (Mann and Szajkó, 2012)
FAP08	238	7660	FAP (Mann and Szajkó, 2012)
FAP09	304	10983	FAP (Mann and Szajkó, 2012)
FAP10	263	8683	FAP (Mann and Szajkó, 2012)
FAP11	285	8016	FAP (Mann and Szajkó, 2012)
FAP12	466	8008	FAP (Mann and Szajkó, 2012)



Then, the main search instance is merged with the search instance 2 and continues from the current location of the search instance 2, see figure below. Afterwards, the main search instance will search through the whole search tree and backtrack to the root node with failure.



7. Empirical results

In a set of empirical measurements using graph coloring benchmarks, we compared the efficiency of the basic backtrack algorithm (BACKTRACK), the backtrack algorithm with frequent restarts (RESTART) as suggested previously in the literature, and the backtrack algorithm with the proposed best-first-search heuristic (BFS).

Table 2. Properties of the random instances used.

Name	n	m
random_65_05_01	65	1063
random_65_05_02	65	1049
random_65_05_03	65	1041
random_70_05_01	70	1203
random_70_05_02	70	1216
random_70_05_03	70	1168
random_70_05_04	70	1166
random_70_05_05	70	1205
random_75_05_01	75	1369
random_75_05_02	75	1410
random_75_05_03	75	1382
random_75_05_04	75	1368
random_75_05_05	75	1375
random_75_052_01	75	1422
random_75_052_02	75	1432
random_75_052_03	75	1446
random_75_052_04	75	1409
random_75_052_05	75	1396
random_80_05_01	80	1575
random_80_05_02	80	1614
random_80_05_03	80	1581
random_80_05_04	80	1590
random_80_05_05	80	1620
random_80_052_01	80	1652
random_80_052_02	80	1659
random_80_052_03	80	1646
random_80_052_04	80	1665
random_80_052_05	80	1637

7.1. Experimental setup.

All measurements were carried out on a computer with an Intel i5 Core CPU running at 2.4 GHz, 3 GB RAM, and Windows 7. The algorithms were implemented in C++ and compiled using `mingw`, the Windows version of GNU GCC 4.4.1. The measurements were carried out using BCAT (Budapest Complexity Analysis Toolkit), a software framework specifically designed for the implementation and testing of combinatorial algorithms (Mann and Szép, 2010).

As problem instances, we used 28 industry benchmarks from different benchmark collections (see Table 1) plus 28 random instances (see Table 2). For all these graphs, we used two different choices for the number of colors k: one such that the graph is k-colorable and one such that it is not. Thus, altogether, we had 112 instances, half of them solvable, the other half unsolvable.

For the RESTART algorithm, we used a geometric restart strategy, the success of which is documented in the literature (Walsh, 1999; Wu and van Beek, 2003): the bound on the number of backtracks before restarting is initialized to 100 and multiplied by 1.5 after each restart. The BFS algorithm is configured so that it starts 500 search instances in depth 15. A switch between solver instances is performed every 8000 backtracks, and in such cases the most attractive solver instance is chosen with probability 0.5 and a random one otherwise.

7.2. Comparing the efficiency of the algorithms.

After fixing the parameters, we ran the three algorithms on all problem instances. Since there were huge differences between the complexity of the different problem instances, we grouped them into three categories based on the runtime of the algorithms on them:

- *Easy* instances with solver runtimes up to 10^1 seconds.
- *Hard* instances with solver runtimes in the range of 10^1 to 10^3 seconds.
- *Hardest* instances with solver runtimes of 10^4 seconds and more.

Table 3. Results on easy problem instances.

	BACKTRACK	RESTART	BFS
Average runtime	3.6 s	6.8 s	6.9 s
Percentage of instances solved within 1 s	69.7%	71.2%	37.9%

The algorithms' results on the easy problem instances are summarized in Table 3. As can be seen, the BACKTRACK algorithm is quite successful here: it solves most problem instances within 1 second, and its average runtime is only 3.6 seconds. RESTART solves slightly more instances within 1 second, but its average runtime is worse because on the "less easy" easy instances its runtime is higher than that of BACKTRACK. BFS has a similar average runtime as RESTART, but the number of instances that it can solve within 1 second is lower. This is probably due to the initial overhead of setting up BFS (several search instances, additional data structures, etc.).

Concerning the hard instances, the algorithms' performance is quite different on solvable vs. unsolvable instances. As discussed, the main motivation for both RESTART and BFS is to be able to give up unfruitful parts of the search space in favor of more promising ones; and this idea applies primarily to solvable instances. For unsolvable instances, in each branching, all possible values must be tried anyway; thus, the systematic search of BACKTRACK is appropriate, restarting or jumping between regions only adds overhead. In principle, restarting could help even in this case because choosing the variables to branch on in a different order may prove better. However, according to the empirical results, it does not help as much as it adds overhead by discarding the results of already performed work.

As can be seen in Fig. 6, both RESTART and BFS help to decrease the runtime on hard solvable instances. In almost all cases, BFS is the fastest. The reduction in

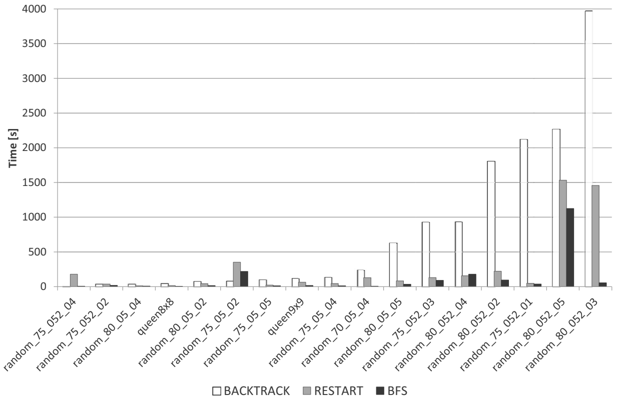


Fig. 6. Results on hard solvable instances.

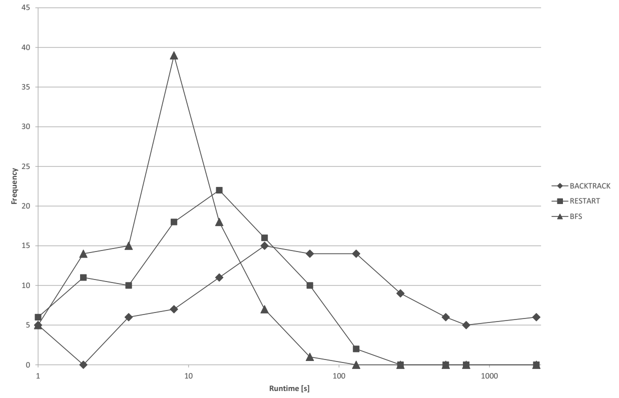


Fig. 8. Histogram of the algorithms' runtime.

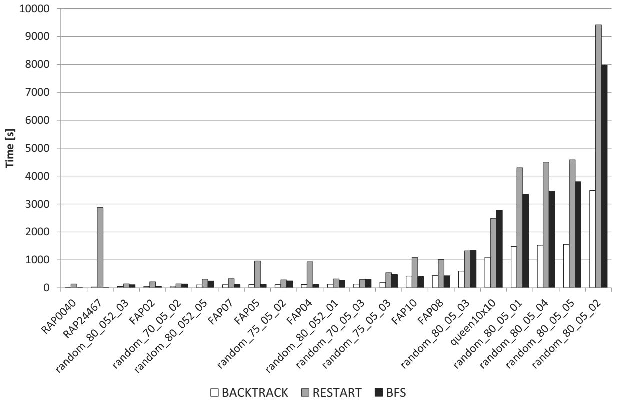


Fig. 7. Results on hard unsolvable instances.

average runtime compared to BACKTRACK is 67% for RESTART and 86% for BFS. In other words, RESTART is approximately 3 times faster than BACKTRACK, whereas BFS is 7 times faster than BACKTRACK.

As expected, BACKTRACK is best for unsolvable instances, as can be seen in Fig. 7. In almost all cases, both RESTART and BFS yield a non-negligible overhead. However, the overhead of BFS is almost always lower than that of RESTART. On average, RESTART is approximately 3 times slower than BACKTRACK. BFS is only 2 times slower than BACKTRACK.

The results on the hardest instances are shown in Table 4. We used a timeout of 8 hours (28800 seconds). As can be seen from the table, BACKTRACK managed to solve 6 out of 7 instances, RESTART solved only 2 of them, whereas BFS solved all 7. The reason why RESTART performed rather poorly on these instances is that most of them are unsolvable, and RESTART adds quite some overhead compared to the other algorithms.

7.3. Runtime variance. Finally, we wanted to assess the variance in the algorithms' runtime. We ran each algorithm 100 times on the same solvable problem instance (random_80_05_02). As can be seen in Fig. 8, BACKTRACK indeed exhibits a heavy-tailed runtime distribution (note the logarithmic scale of the horizontal axis). Both RESTART and BFS have far lower tail probabilities. In particular, BFS has a sharp concentration around its maximum likelihood value at approximately 10 s. In other words: in terms of both efficiency and predictability of runtime, RESTART is superior to BACKTRACK, and BFS is even superior to RESTART.

8. Conclusions

In this paper, we proposed a new way to reduce the runtime of backtrack search. We observed that the reason why backtrack search lasts sometimes extraordinarily long is caused by its depth-first-search nature prohibiting it from giving up on unfruitful parts of the search space in favor of more promising areas. To remedy this, we proposed to extend backtrack search with a best-first-search control, thus combining the systematic backtrack search with exploration options. Our algorithm operates multiple backtrack search instances and focuses always on the most promising one. We described in detail how this scheme can be implemented efficiently and without sacrificing optimality.

Our work is related to the frequent restart strategy that had been suggested previously in the literature with a similar goal. The advantage of our approach over restarting is that we do not discard knowledge cumulated by the backtrack search. Our empirical results confirmed that our algorithm is indeed more efficient than restarting: it results in higher speedup on solvable problem instances and lower overhead on unsolvable instances.

Although the results are promising, there is still work to be done. We regard the complexity of our approach as the main obstacle to its wide-spread usage. Although the basic idea of our algorithm is not complicated, implementing it on top of an existing solver can be tricky and may depend on the characteristics of the solver, as

Table 4. Runtimes (in sec.) on the hardest problem instances.

Benchmark	Solvable	BACKTRACK	RESTART	BFS
FAP01	no	4864	timeout	4932
FAP09	no	7373	24033	7390
queen9x9	no	10149	timeout	10542
queen10x10	yes	timeout	timeout	17995
random_80_052_01	yes	11872	25789	18056
RAP24050	no	4145	timeout	4339
RAP1792	no	13601	timeout	14657

shown also in this paper in relation to conflict-driven backjumping. In the future, we would like to gain more experience with the applicability and implementation options of this approach. In particular, we intend to integrate it into a state-of-the-art satisfiability solver.

Another promising avenue for future research is the parallelization of the presented approach. Since the work of the individual search instances is mostly independent of each other, with only limited interaction at well-defined points, we can expect a significant performance gain from parallelization. On a system with k parallel processing units, one can run k worker threads that process search instances in parallel, and a manager thread that determines the next search instance to process for a worker thread that has become free, based on the best-first-search logic. This way, an almost k-fold speedup can be expected.

Acknowledgment

This work was partially supported by the Hungarian Scientific Research Fund (Grant No. OTKA 108947) and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

Appel, A.W. and George, L. (1996). Register interference graphs, `http://www.cs.princeton.edu/~appel/graphdata/`.

Bender, E.A. and Wilf, H.S. (1985). A theoretical analysis of backtracking in the graph coloring problem, *Journal of Algorithms* **6**(2): 275–282.

Biere, A. (2008). Adaptive restart strategies for conflict driven SAT solvers, *in* H. Kleine Büning and X. Zhao (Eds.), *Theory and Applications of Satisfiability Testing—SAT 2008*, Springer, Berlin, pp. 28–33.

Brélaz, D. (1979). New methods to color the vertices of a graph, *Communications of the ACM* **22**(4): 251–256.

Brown, C.A., Finkelstein, L. and Purdom, P.W.J. (1996). Backtrack searching in the presence of symmetry, *Nordic Journal of Computing* **3**(3): 203–219.

Brown, J.R. (1972). Chromatic scheduling and the chromatic number problem, *Management Science* **19**(4): 456–463.

Cheeseman, P., Kanefsky, B. and Taylor, W.M. (1991). Where the really hard problems are, *12th International Joint Conference on Artificial Intelligence (IJCAI '91), Sydney, Australia*, pp. 331–337.

Davis, M., Logemann, G. and Loveland, D. (1962). A machine program for theorem proving, *Communications of the ACM* **5**(7): 394–397.

Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory, *Journal of the ACM* **7**(3): 201–215.

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition, *Artificial Intelligence* **41**(3): 273–312.

Dechter, R. (2003). *Constraint Processing*, Morgan Kaufmann, San Francisco, CA.

Dechter, R. and Frost, D. (2002). Backjump-based backtracking for constraint satisfaction problems, *Artificial Intelligence* **136**(2): 147–188.

Eén, N. and Sörensson, N. (2004). An extensible SAT-solver, *in* E. Giunchiglia and A. Tacchella (Eds.), *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, Vol. 2919, Springer, Berlin/Heidelberg, pp. 502–518.

Geelen, P.A. (1992). Dual viewpoint heuristics for binary constraint satisfaction problems, *Proceedings of the 10th European Conference on Artificial Intelligence, Vienna, Austria*, pp. 31–35.

Gomes, C.P., Selman, B., Crato, N. and Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, *Journal of Automated Reasoning* **24**(1–2): 67–100.

Gomes, C., Selman, B. and Kautz, H. (1998). Boosting combinatorial search through randomization, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI, USA*, pp. 431–437.

Haim, S. and Heule, M. (2010). Towards ultra rapid restarts, *Technical report*, UNSW/TU Delft, Sydney/Delft.

Hogg, T. and Williams, C.P. (1994). The hardest constraint problems: A double phase transition, *Artificial Intelligence* **69**(1–2): 359–377.

Hutter, F., Hamadi, Y., Hoos, H. and Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms, *in* F. Benhamou (Ed.), *Principles and Practice of Constraint Programming—CP 2006*, Springer, Berlin/Heidelberg, pp. 213–228.

Jia, H. and Moore, C. (2004). How much backtracking does it take to color random graphs? Rigorous results on heavy tails, *Principles and Practice of Constraint Programming (CP 2004), Toronto, Canada*, pp. 742–746.

Kautz, H., Horvitz, E., Ruan, Y., Gomes, C. and Selman, B. (2002). Dynamic restart policies, *18th National Conference on Artificial Intelligence, Edmonton, Canada*, pp. 674–681.

Knuth, D.E. (1975). Estimating the efficiency of backtrack programs, *Mathematics of Computation* **29**(129): 121–139.

Luby, M., Sinclair, A. and Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms, *Information Processing Letters* **47**(4): 173–180.

Mann, Z. (2011). *Optimization in Computer Engineering—Theory and Applications*, Scientific Research Publishing, Irvine, CA.

Mann, Z. and Szajkó, A. (2012). Complexity of different ILP models of the frequency assignment problem, *in* Z. Mann (Ed.), *Linear Programming—New Frontiers in Theory and Applications*, Nova Science Publishers, New York, NY, pp. 305–326.

Mann, Z. and Szajkó, A. (2010a). Determining the expected runtime of exact graph coloring, *Proceedings of the 13th International Multiconference on Information Society—IS 2010, Ljubljana, Slovenia*, Vol. A, pp. 389–393.

Mann, Z. and Szajkó, A. (2010b). Improved bounds on the complexity of graph coloring, *Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania*, pp. 347–354.

Mann, Z. and Szép, T. (2010). BCAT: A framework for analyzing the complexity of algorithms, *8th IEEE International Symposium on Intelligent Systems and Informatics, Subotica, Serbia*, pp. 297–302.

Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. and Malik, S. (2001). Chaff: Engineering an efficient SAT solver, *Proceedings of the 38th Annual Design Automation Conference, Las Vegas, NV, USA*, pp. 530–535.

Russell, S.J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*, 3rd Edn., Prentice Hall, Upper Saddle River, NJ.

Schaefer, R., Byrski, A., Kołodziej, J. and Smołka, M. (2012). An agent-based model of hierarchic genetic search, *Computers and Mathematics with Applications* **64**(12): 3763–3776.

Trick, M. (2003). COLOR03 graph coloring benchmarks, `http://mat.gsia.cmu.edu/COLOR03/`.

Walsh, T. (1999). Search in a small world, *Proceedings of the 16th International Joint Conference on Artificial Intelligence, Stockholm, Sweden*, pp. 1172–1177.

Wilf, H.S. (1984). Backtrack: An O(1) expected time algorithm for the graph coloring problem, *Information Processing Letters* **18**(3): 119–121.

Wu, H. and van Beek, P. (2003). Restart strategies: Analysis and simulation, *Principles and Practice of Constraint Programming—CP 2003, Kinsale, Ireland*, p. 1001.

Zoltán Ádám Mann received his M.Sc. and Ph.D. degrees in computer science from the Budapest University of Technology and Economics in 2001 and 2005, respectively. He also received an M.Sc. degree in mathematics from Eötvös Loránd University in 2004. Currently, he is an associate professor at the Department of Computer Science and Information Theory, Budapest University of Technology and Economics. His main research interests are combinatorial algorithms and algorithmic complexity.

Tamás Szép received the M.Sc. degree in computer science from the Budapest University of Technology and Economics in 2014. Currently, he commences his Ph.D. studies at the Karlsruhe Institute of Technology. His main areas of research interest are algorithm theory and computer graphics.

Received: 2 February 2014
Revised: 7 May 2014